OpenSolaris Scheduling and CPU Management

Eric Saxe (eric dot saxe at sun dot com)
Solaris Core Kernel Development
http://blogs.sun.com/esaxe

Introduction

- Processor / system architectures becoming increasingly complex...
  - Chip Multi-threaded processors (CMT): multi-core, multi-threaded, shared caches...
  - Non-Uniform Memory Access systems (NUMA)
- Soon, you won't be able to purchase a “uni-processor” system
- How does OpenSolaris utilize these increasingly complex processors?
- How can / should you manage them?

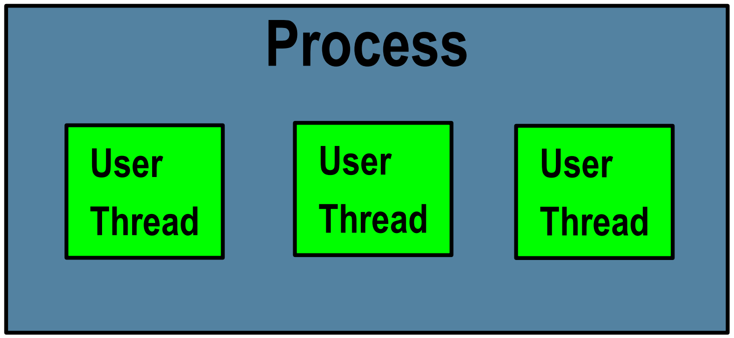
Outline

- Processes, LWPs, and Threads
- Dispatcher Overview, Scheduling Classes
- Processor Abstractions, Tools, and Interfaces
- “Under the hood” with mdb(1), dtrace(1m)
- Looking ahead

Processes, LWPs, and Threads

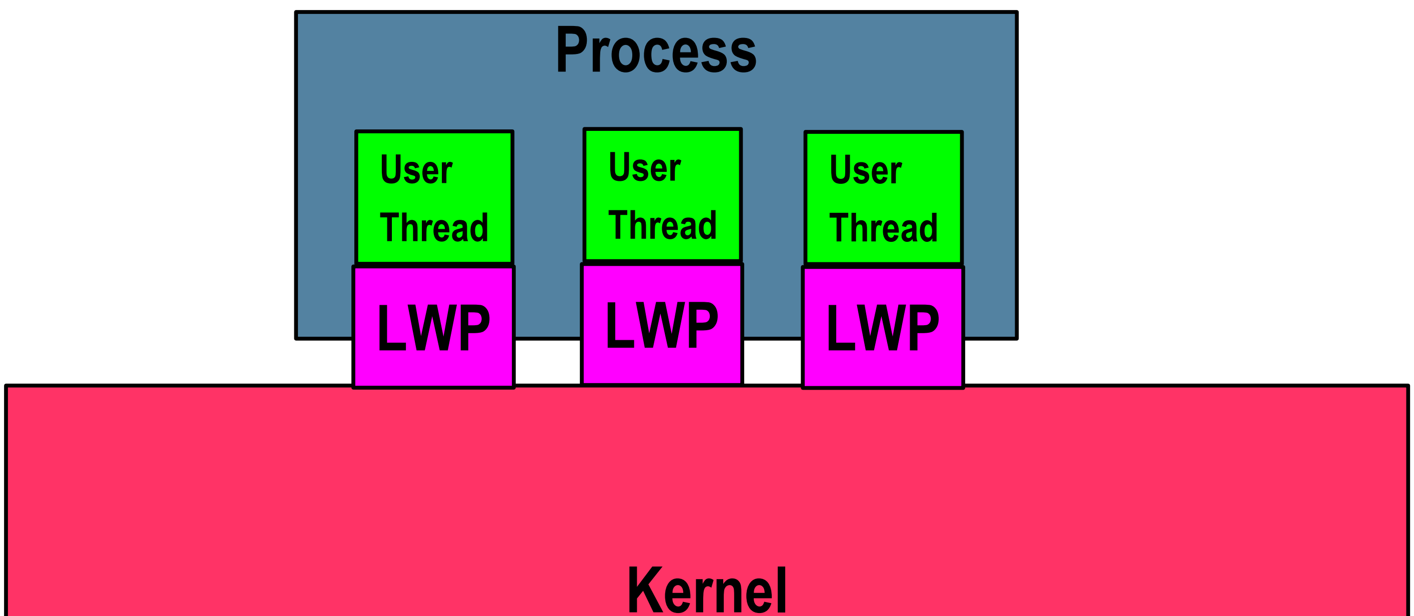
Processes, LWPs, Threads

- Process: “container” for an executable object
  - > Has an associated VM address space
  - > ...and one or more threads of execution (that share the address space)
  - > proc_t defined in *uts/common/sys/proc.h*



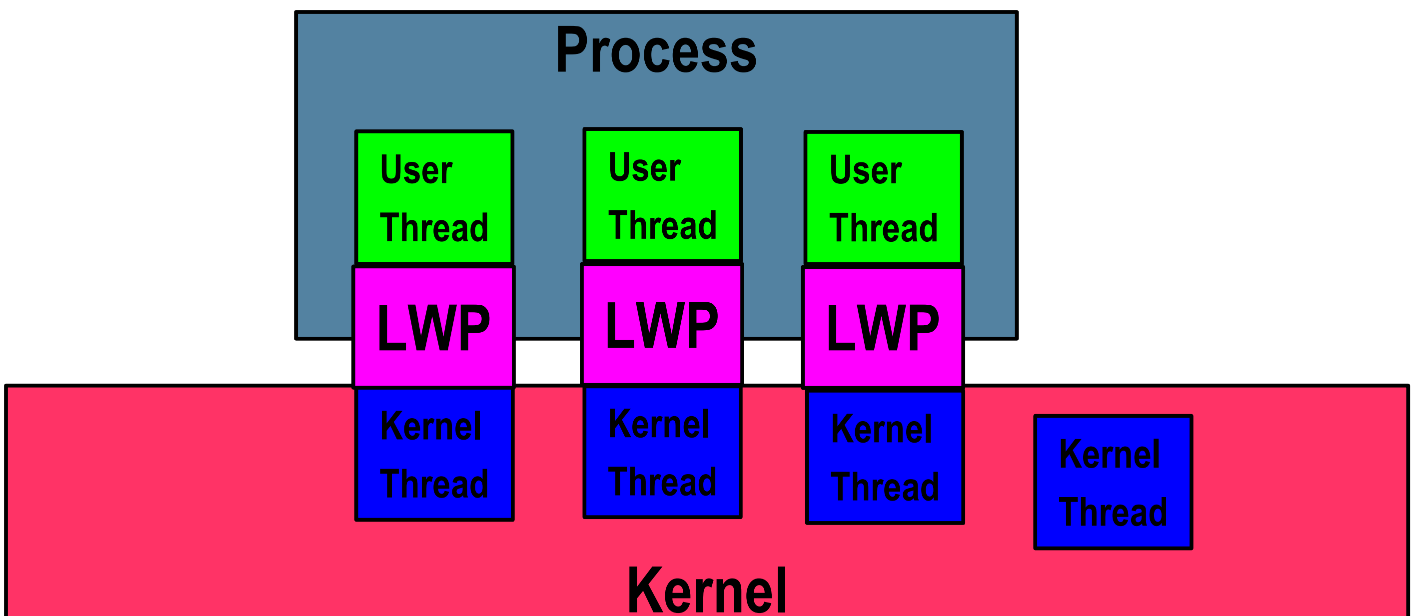
Processes, LWPs, Threads...

- Each thread in a process has an associated LWP (Light Weight Process)...a kernel object that maintains a user thread's state:
  - System call / signal info, accounting, debugger state, ...
- klwp_t defined in *uts/common/sys/klwp.h*



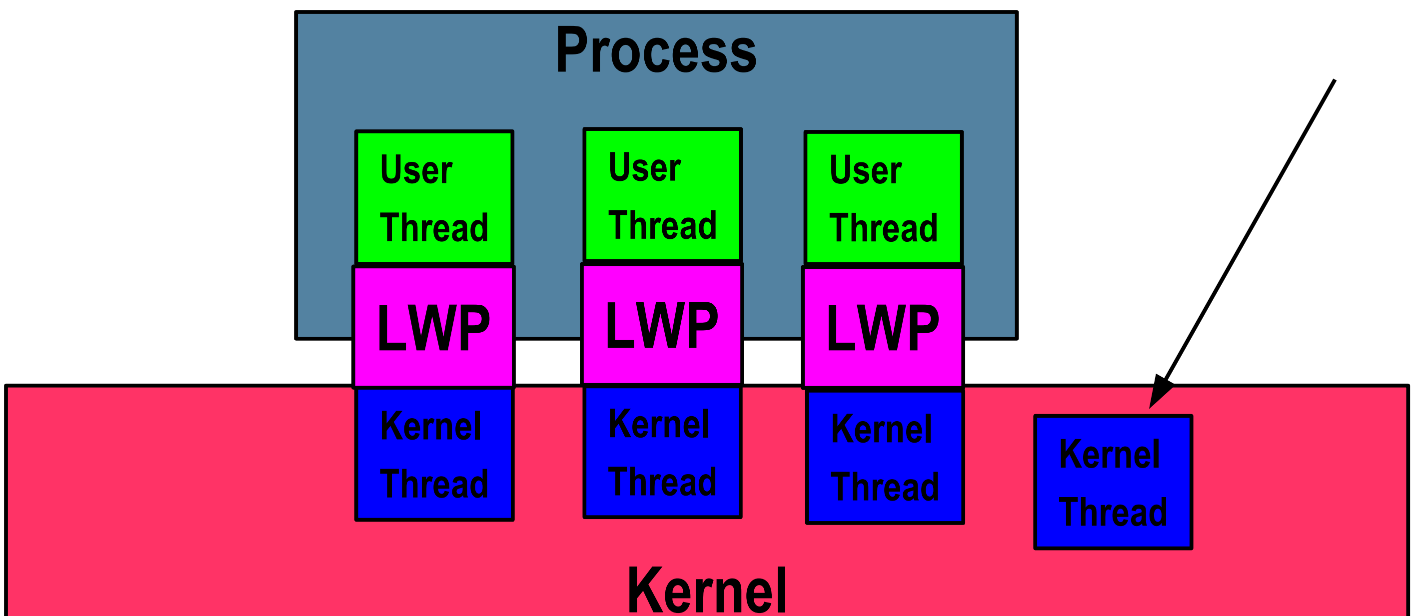
Processes, LWPs, Threads...

- Linked to each LWP is a kernel thread
- The kernel thread is the fundamental unit of scheduling and execution in the system
  - kthread_t defined in *uts/common/sys/thread.h*



Processes, LWPs, Threads...

- Some kernel threads may not have an associated LWP
- These are kernel service threads
  - > Examples: CPU Idle threads, task queue worker threads, ...



Thread States

- At any given time, a (k)thread is either:
  - Runnable: ready to run, but not running
  - “On Proc”: running on a CPU
  - Sleeping: blocked waiting for something
- Less frequently, a thread may also be
  - Zombied: exited (dead), but not yet reaped
  - Free: exited (dead) and reaped
  - Stopped: Suspended (initial creation / pstop())
- States defined in uts/common/sys/thread.h

Some Process tools...

- prstat(1m) - "top" like tool
- /proc tools
  - > pstop(1) – stop a process
  - > prun(1) – opposite of pstop(1)
  - > pstack(1) – show stack traces for processes LWPs
  - > pcred(1) – show / set credentials
  - > pfiles(1) – report open files
  - > See proc(1) for more...

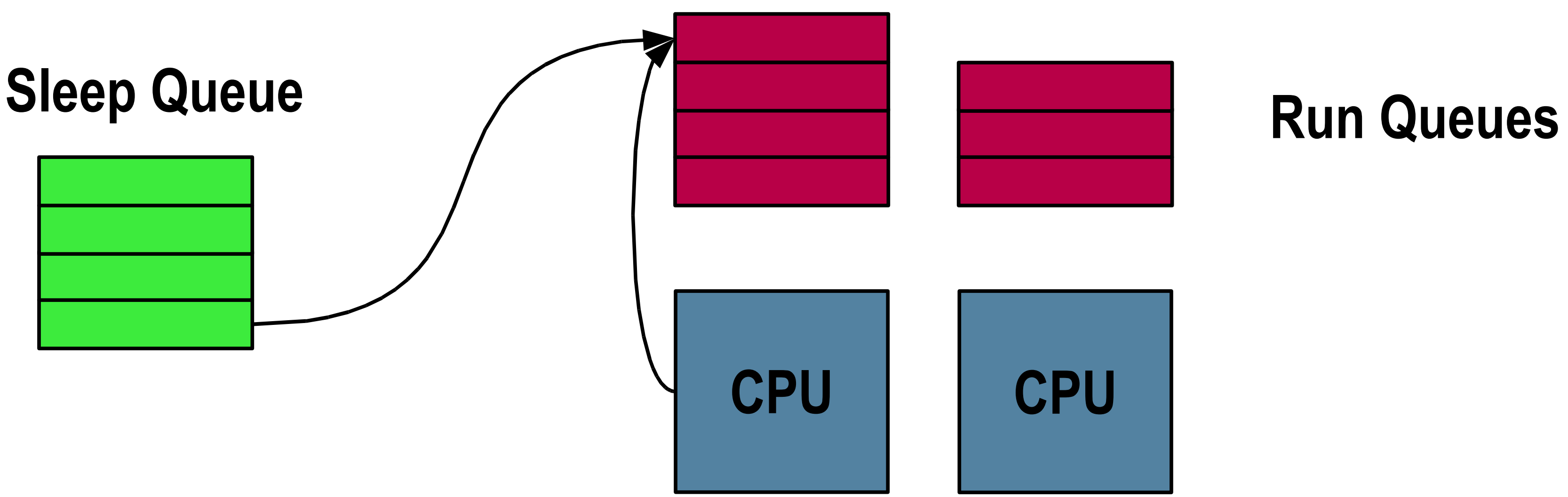
Scheduling Classes and the Dispatcher

Dispatcher/Scheduler

- The dispatcher is the kernel subsystem that decides **where** (on which CPUs) runnable threads should be scheduled to run.
- Threads will be in one of several scheduling classes who's policies dictate **when** the thread will run (by managing the thread's priority) with respect to other threads.
- Threads enter the dispatcher when making transitions (or when causing other threads to transition) between thread states.

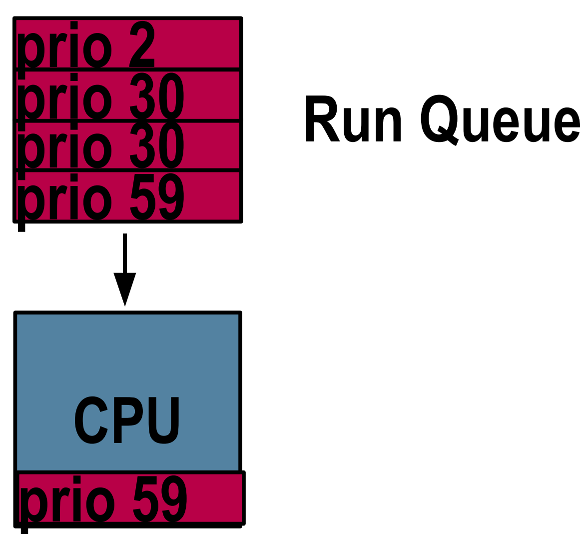
Dispatcher and Thread States

- {Sleep, On Proc} => Runnable
  - The dispatcher is entered where it chooses a CPU, and then enqueues the thread on that CPU's run queue



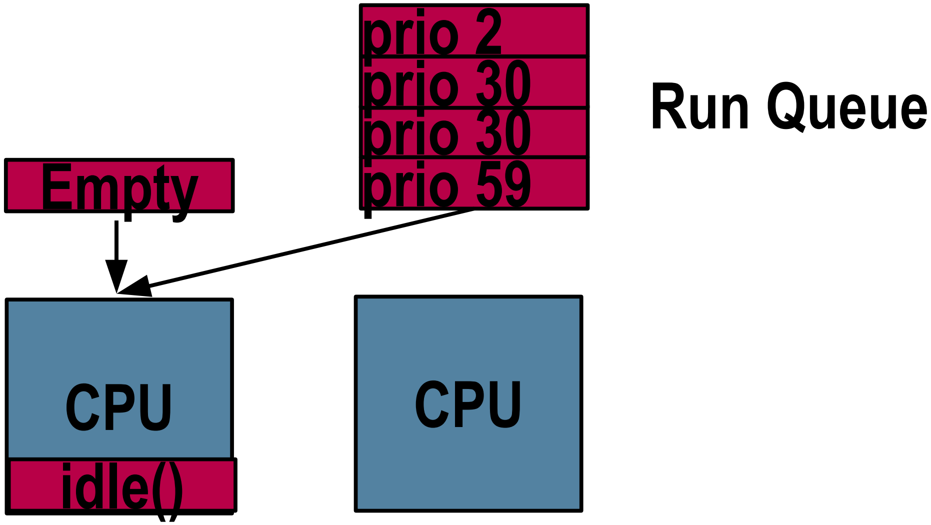
Dispatcher and Thread States

- Runnable => On Proc
  - When the currently executing thread surrenders the CPU, the dispatcher is entered and the highest priority thread on the CPU's queue is dequeued and (context) switched to.



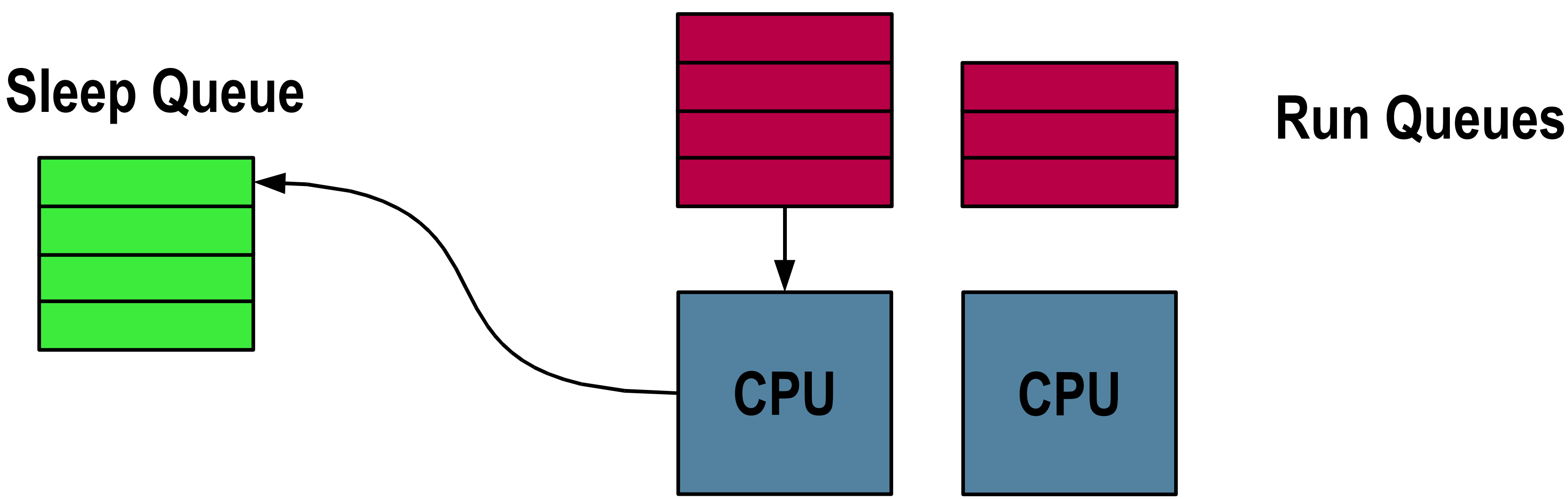
Dispatcher and Thread States

- Runnable => On Proc
  - If that CPU's queue is empty, the dispatcher switches in the CPU's “idle” thread...which trolls around the other CPU's run queues looking for work to steal.



Dispatcher and Thread States

- On Proc => Sleep
  - The blocking thread surrenders the CPU, and enqueues itself on the synchronization object's sleep queue. It then pulls the highest priority thread from the run queue, and switches to it.



Putting it together...

- Running thread finishes it's time slice...
  - clock(), while doing tick accounting for the thread, realizes that the thread's time slice is up..
  - the running thread is preempted
    - cpu_runrun flag set on running thread's CPU structure, and a cross trap is sent
    - running thread traps, and sees cpu_runrun. It then calls preempt()
  - In preempt() the thread enters the dispatcher, to find a CPU on which to schedule itself to run
  - After enqueueing itself, it calls swtch()...which context switches to the highest priority thread waiting in the CPU's run queue

Putting it together...

- Running thread (A) drops a lock for which another thread (B) is waiting
  - (A) dropping the lock finds the sleep queue associated with the lock, and finds (B) sleeping
  - (A) dequeues (B) from the sleep queue, and enters the dispatcher to schedule now runnable thread (B)
  - In the dispatcher (A) enqueues (B) in an appropriate CPU's run queue
  - (A) continues running
  - (B) remains runnable in the run queue, waiting to be put “on proc”

Scheduling Classes

- Time Share (TS) class
  - Operates over global priority range: 0-59
  - Priority adjustments made based on how long threads spend using (vs waiting for) processor resources
  - CPU bound => priority drops
  - Interactive => priority increases
- Interactive (IA) class
  - Operates over global priority range: 0-59
  - Like TS, but with an added priority “boost” mechanism
  - Used to improve interactivity of “in focus/use” processes
    - Xserver, etc

Scheduling Classes

- Fair Share (FSS) class
  - > Operates over global priority range: 0-59
  - > Processor resources provisioned into “shares” assigned to processes managed by the Solaris resource management facility
  - > Priority adjusted according to share allocation and relative processor utilization
- Fixed Priority (FX) class
  - > Operates over global priority range: 0-60
  - > Priorities are static. Privileges needed to enter at priorities greater than 0

Scheduling Classes

- System (SYS) class
  - > Operates over global priority range: 60-99
  - > Used by kernel service threads
- Real Time (RT) class
  - > Operates over global priority range: 100-159
  - > When fastest possible dispatch latencies are required...
  - > RT threads can preempt the kernel
  - > Use with caution

Using Scheduling Classes

- priocntl(1) used to change the scheduling class and priority of new or existing threads
  - Example: Move the shell (and anything it invokes) into the RT scheduling class
    - `# priocntl -s -c RT -i pid $$`
- dispadmin(1M) used to get (and set) scheduling class parameters on the fly

```
esaxe@jet$ dispadmin -g -c TS
# Time Sharing Dispatcher Configuration
RES=1000

# ts_quantum  ts_tqexp  ts_slpret  ts_maxwait ts_lwait  PRIORITY LEVEL
       200         0        50           0        50        #     0
       200         0        50           0        50        #     1
       200         0        50           0        50        #     2
       200         0        50           0        50        #     3
...
```

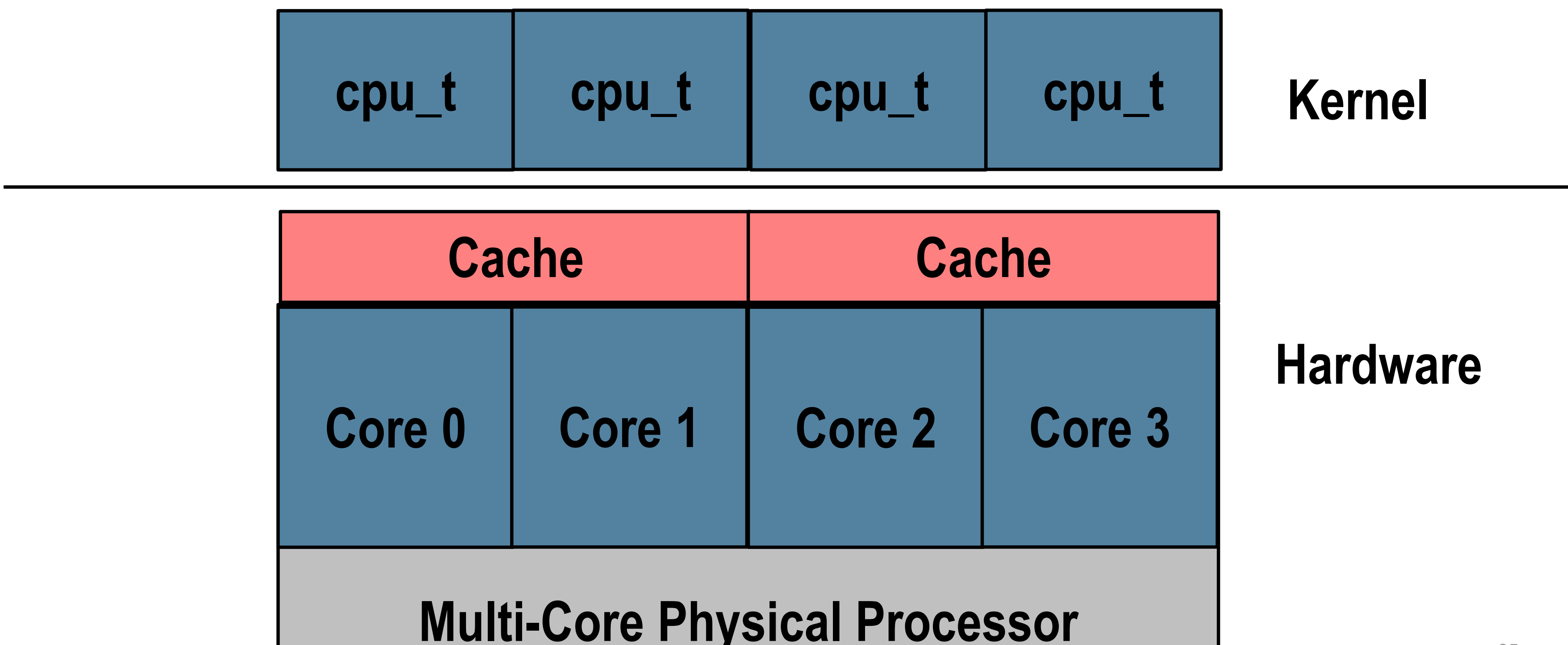
Processor Related Abstractions

The “logical” processor: cpu_t

- The CPU, a.k.a. “struct cpu” or cpu_t is the kernel's fundamental processor abstraction, representing a execution resource capable of running one thread of execution at a time.
- Traditional processors present to the OS a single logical processor, or CPU.
- Today's multi-threaded, multi-core (CMT) processors present multiple logical processors, as they are capable of running multiple threads simultaneously.

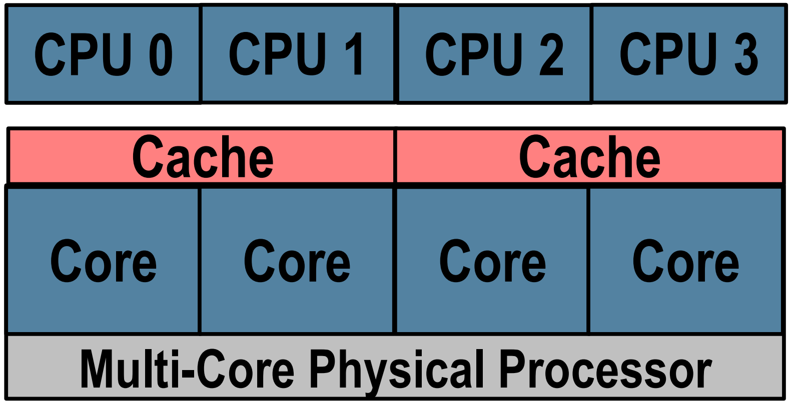
CMT processors & “logical” CPUs

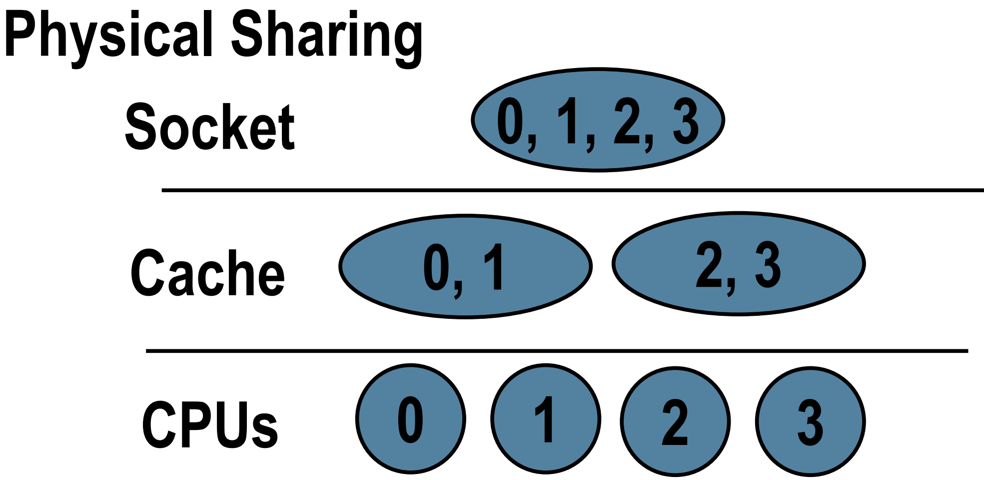
- The multiple logical CPUs presented may share physical processor components / resources...



Processor Group Abstraction

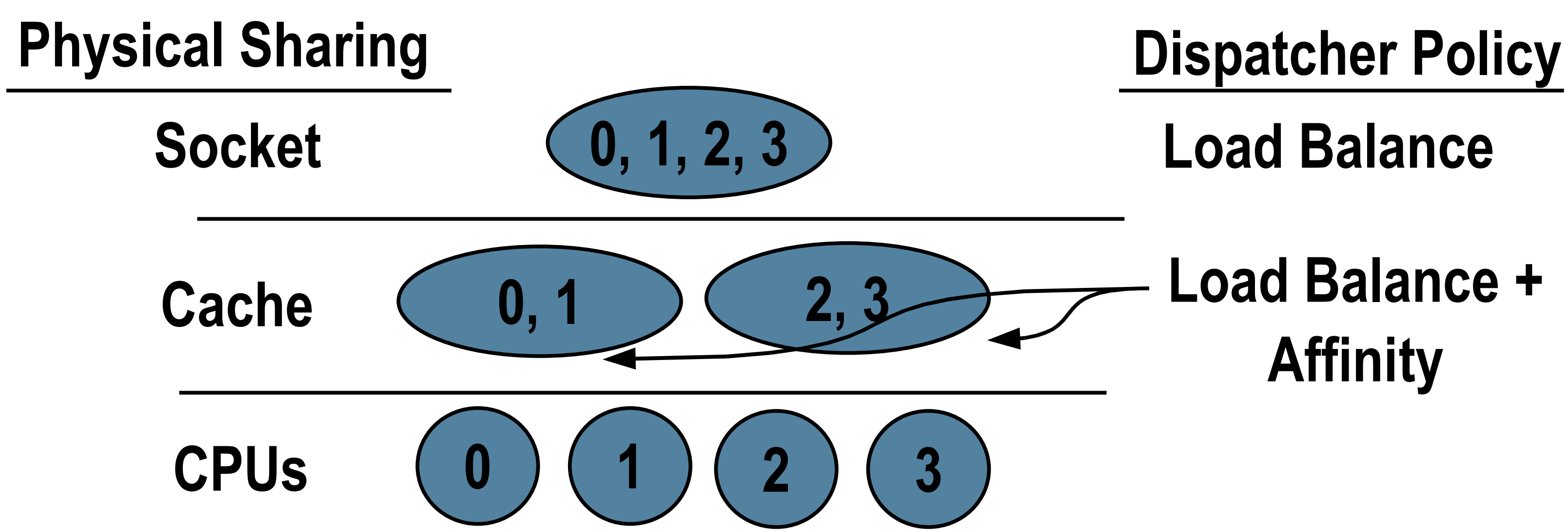
- The kernel detects CMT sharing relationships existing between logical CPUs which it represents though a hierarchy of “processor groups”.
- The “processor group” (pg_t) kernel abstraction represents a group of CPUs with some physical or characteristics sharing relationship.





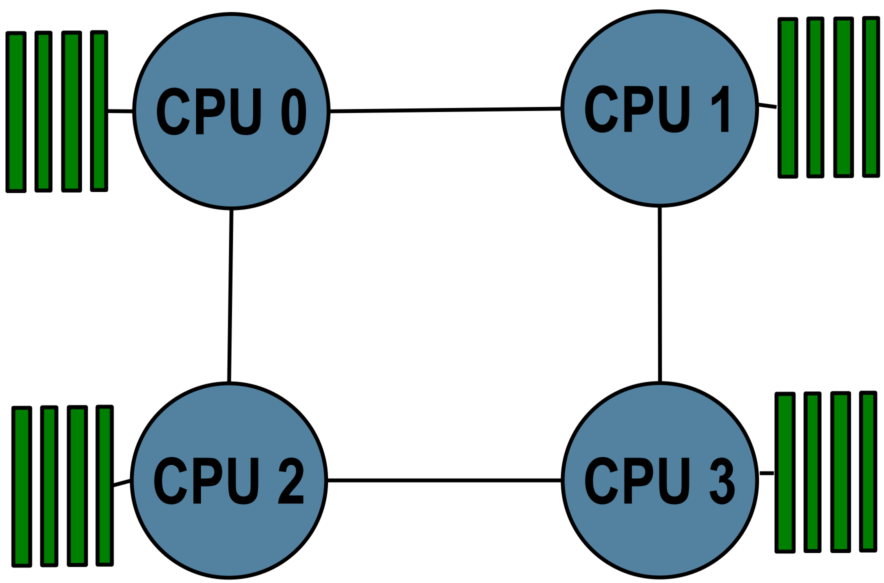
Processor Group Abstraction

- The dispatcher consults these groupings to implement load balancing and affinity scheduling policy that optimize for the nuances of the hardware.



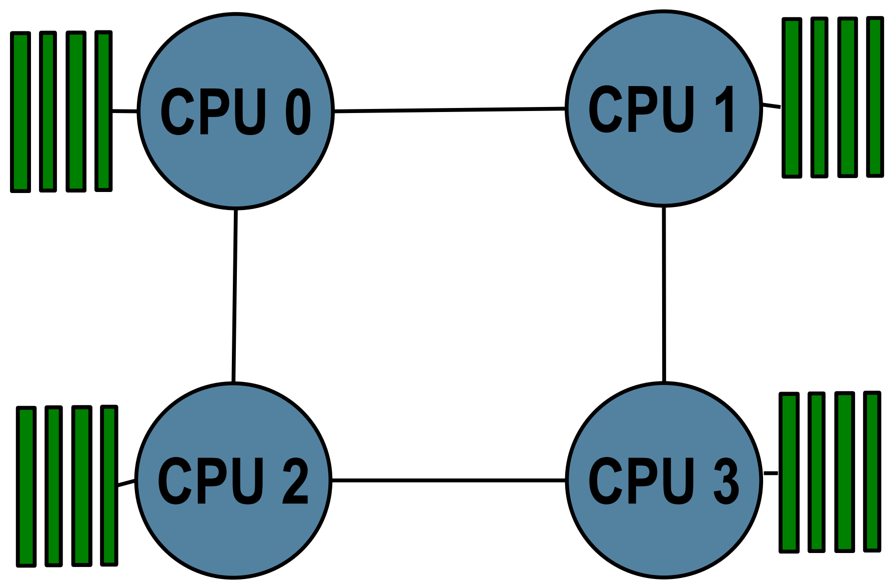
lgroups

- On systems having Non-Uniform Memory Access (NUMA) architectures, some physical memory is close, while other memory is “farther away” (from a given CPU's perspective)...



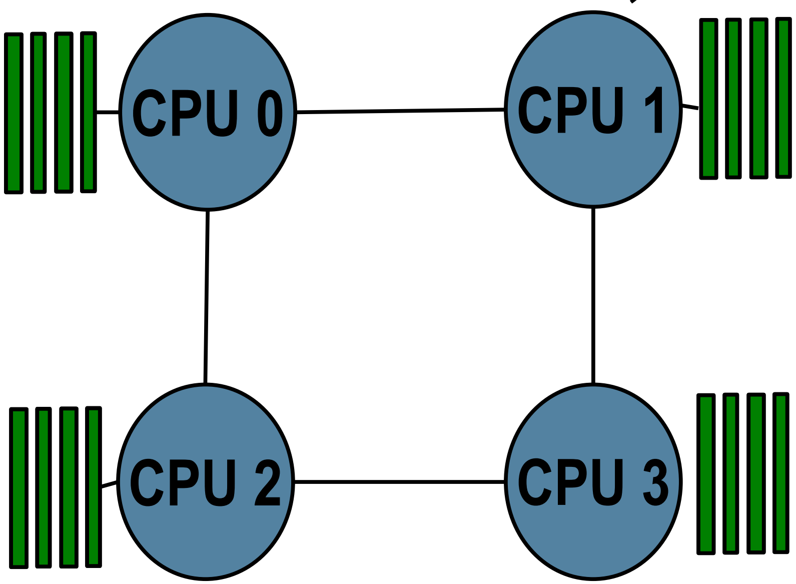
lgroups

- A “locality group” (lgroup) abstraction represents a group of CPU and memory resources that are within some latency of each other.

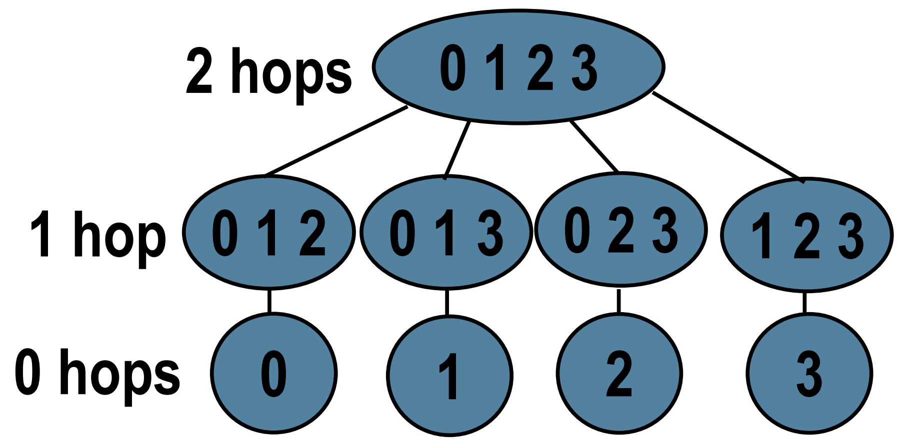


lgroups

- This topology has 3 levels of locality...
- The kernel arranges the lgroups it creates into a hierarchy to make it easy to find the closest, the next closest, ... resources.



Topology



lgroup hierarchy

lgroups

- Each thread in the system is assigned a “home” lgroup.
  > The dispatcher tries to run the thread in (or as near as possible to it's home)
  > Likewise, the VM subsystem tries to allocate memory close to the thread's home.
- Result:
  > threads tend to run near the memory they've allocated.
  > Average incurred memory latency is minimized, and performance improves.

Processor Sets

- Not to be confused with Processor Groups...
- A processor set is a user created set of CPUs
  - > Threads must be "bound" to the pset, to run on any of the CPUs in that set.
  - > Once bound, threads cannot run on CPUs outside the set.
- This is useful for provisioning CPU resources for various workloads on the system, as well as "fencing off" workloads from one another.
- psets are administered via the psrset(1M) command

CPU and Processor Set Tools

- mpstat(1M) – report per CPU statistics
- pbind(1M) – bind thread(s) to the specified CPU
- psradm(1M) – change the state of the specified CPU
  - online, offline, no_intr, ...
- psrinfo(1M) – displays CPU information
  - psrinfo -vp option added to provide limited physical view
- psrset(1M) – administer processor sets

Processor Related Interfaces

- CPU:
  - p_online(2) – Change CPU states
  - processor_bind(2) – Bind LWPs to a processor
  - processor_info(2) – Query type / status of a processor
- Processor Sets:
  - pset_create(2), pset_destroy(2), pset_assign(2)
    - create, destroy, and assign CPUs to processor sets
  - pset_bind(2) – bind LWPs to a processor set
- In development:
  - Multi-CPU binding – Bind to a set of CPUs
    - Like processor sets, but non-exclusive

Tools and Interfaces: NUMA

- Tools:
  - > plgrp(1) – Set / get a thread's home lgroup
  - > lgrpinfo(1) – Display information about system's locality groups, and the lgroup hierarchy
  - > pmadvise(1) – Provide “advice” about usage for a given range of virtual memory.
    - > On NUMA systems, the kernel will migrate pages to improve locality
  - > pmap(1) – Using “-L” option, show where (in which lgroups) a process's physical memory resides
- Interfaces:
  - > liblgrp(3LIB)

Under the hood with mdb(1) and dtrace(1M)

mdb(1) debugger commands

- ::cpuinfo -v shows what's running on the system's CPUs, and who's runnable...

```
> ::cpuinfo -v
 ID ADDR             FLG NRUN BSPL PRI RNRN KRNRN SWITCH THREAD           PROC
  0 fffffffffbc26f30  1b    0    0  20   no    no t-0    ffffffec8a58f20 bash
                       |
            RUNNING <--+
              READY
             EXISTS
             ENABLE

 ID ADDR             FLG NRUN BSPL PRI RNRN KRNRN SWITCH THREAD           PROC
  1 ffffffffec15e2800 1b    2    0  59   no    no t-0    ffffffec967e020 mdb
                       |    |
            RUNNING <--+    +-->  PRI THREAD           PROC
              READY                59 ffffffec1ca67e0 gnome-terminal
             EXISTS                59 ffffffec1c9eb60 Xorg
             ENABLE
```

mdb(1) debugger commands

- Use ::findstack to look at the stack for a given thread.
  - Note, this is the stack for the kernel thread, not the stack for the user application
    - get that via pstack(1)

```
> ffffffffec1c9eb60::findstack
stack pointer for thread ffffffffec1c9eb60: ffffff00042f6c40
[ ffffff00042f6c40 _resume_from_idle+0xf8() ]
  ffffff00042f6c80 swtch+0x17f()
  ffffff00042f6d10 cv_timedwait_sig+0x194()
  ffffff00042f6da0 cv_waituntil_sig+0xbb()
  ffffff00042f6e80 poll_common+0x3dd()
  ffffff00042f6f00 pollsys+0xec()
  ffffff00042f6f10 sys_syscall+0x17b()
```

mdb(1) debugger commands

- ::ps gives the kernel's view of processes
- Using the address of the proc_t structure, it's easy to “walk” the process's kthread_t structures...

```
> ::ps
S    PID   PPID   PGID    SID    UID      FLAGS             ADDR NAME
R      0      0      0      0      0 0x00000001 fffffffffbc24eb0 sched
R      3      0      0      0      0 0x00020001 ffffffffec1b5ca18 fsflush
R      2      0      0      0      0 0x00020001 ffffffffec1b5d670 pageout
R      1      0      0      0      0 0x4a004000 ffffffffec1b5e2c8 init
R   1094   1062    885    885  90119 0x4a004000 ffffffffec4361030 soffice.bin
...
> ffffffffec4361030::walk thread
ffffffffec880bf00
ffffffffec8a595e0
ffffffffec1dfae20
ffffffffec1d0f920
ffffffffec1d07ca0
ffffffffec880c260
```

mdb(1) debugger commands

- “pipe” the walk output to other interesting debugger commands, like ::print...

```
> ffffffffec4361030::walk thread |::print kthread_t t_start
t_start = 2007 Jul 26 11:03:04
t_start = 2007 Jul 26 11:03:04
t_start = 2007 Jul 26 11:03:07
t_start = 2007 Jul 26 11:03:07
t_start = 2007 Jul 26 11:03:07
t_start = 2007 Jul 26 11:03:11
```

The DTrace sched provider

- The sched provider exports a number of interesting scheduling related DTrace probes...
  - > enqueue, dequeue
    - > Fires when a thread is added / removed from a run queue
  - > on-cpu, off-cpu
    - > Fire when a thread gets on, or leaves the CPU
  - > sleep, wakeup
  - > preempt
  - > tick
- See the DTrace answerbook for complete list

DTrace sched provider example

- When firefox gets the CPU, how long does it run?

```
#!/usr/sbin/dtrace -s

sched:::on-cpu
/execname == "firefox-bin"/
{
        self->ts = timestamp;
}

sched:::off-cpu
/self->ts/
{
        @["firefox run times"] = quantize(timestamp - self->ts);
        self->ts = 0;
}
```

DTrace sched provider example

```
# ./ff_howlong.d
dtrace: script './ff_howlong.d' matched 6 probes
^C

  firefox run times
           value  ------------- Distribution ------------- count
            1024 |                                         0
            2048 |@                                        42
            4096 |@@@@@                                    165
            8192 |@@@@@@@@@@@@                             365
           16384 |@@                                       55
           32768 |@@@@@                                    150
           65536 |@@@                                      104
          131072 |@@@                                      90
          262144 |@@                                       48
          524288 |@@                                       52
         1048576 |@@@@                                     132
         2097152 |                                         13
         4194304 |                                         0
```

Looking ahead...

In Development work...

- Tesla Project: OpenSolaris Enhanced Power Management
  - > http://www.opensolaris.org/os/project/tesla
- Short term objectives:
  - > Power aware dispatcher
    - > Make the dispatcher aware of CPU power states
    - > Better integrate the dispatcher with CPU PM subsystem
  - > Event based clock implementation
    - > Currently, clock() interrupt fires 100 times per second, even on completely idle system.
      - – Bad from a power efficiency perspective
    - > clock shouldn't fire unless there's something to do

Future work...

- OpenSolaris CPU Observability Project
  - > Exporting CMT sharing relationships that exist between logical CPUs
  - > Project recently proposed
- Workload characterization, self tuning, and adaptive policies
- CPU related observability / control tools rework...
  - > mpstat(1m)... so much output, so little xterm.

Eric Saxe <eric dot saxe at sun dot com>
http://blogs.sun.com/esaxe