

Intel Corporation

# mOS v0.9 Scheduler

Document Revision: v0.9.3

Attinella, John E  
5-23-2022

## Preface

The mOS (multi Operating System) is an operating system targeting the High Performance Computing (HPC) environment. The mOS is essentially two kernels, one a modified Linux and the other a custom light-weight kernel, co-operating in a single memory coherent computing environment. The hardware resources are shared by these two kernels. The primary components within the light-weight kernel are the memory component and the scheduler component. This document describes the design and operation of the scheduler component.

## Contents

Preface .....	1
Revision History .....	7
Glossary.....	8
Introduction .....	10
mOS Scheduler Attributes.....	10
Low noise, predictive scheduling .....	10
Fast function adaptation.....	10
Behaviors optimized for HPC and AI .....	10
Topology aware thread placement.....	10
Flexible and simple handling of utility threads.....	10
Strict CPU affinity .....	11
Support of pthread libraries and runtimes .....	11
Inter-operability with the Linux scheduler .....	11
Compatible with many existing Linux debug/performance tools.....	11
Dynamically configure CPU resources .....	11
Internal options for experimentation .....	11
Design.....	11
CPU Assignment.....	11
Node Level .....	11
lwkcpus kernel boot parameter.....	12
lwkctl command.....	12
Node CPU configuration diagram .....	13
Process Level.....	13
Process CPU reservation diagram .....	14
Potential Conflicts .....	14
Thread Level.....	14
Thread placement example .....	15
Utility Threads.....	15
Utility Thread API .....	15
Heuristic utility thread placement .....	16
Running on a Linux CPU .....	17
Over-commitment .....	17

More compute threads than CPUs .....	18
Explicit compute thread placement.....	18
Wake-up balancing .....	18
Thread placement through Utility Thread API.....	18
Compute and Utility thread interactions .....	18
Thread Push/Pull Balancers .....	19
Pull Balancer.....	19
Push Balancer.....	20
CPU Isolation and configuration .....	21
Linux CPU hot-plug.....	22
Linux modifications and mOS Scheduler code.....	22
Modified Linux Source Files .....	23
New mOS Scheduler Source Files .....	23
Correctable Machine Checks .....	23
Clock source watchdog .....	24
GPU Resource Management.....	25
GPU User Interfaces.....	25
GPU Affinity Control.....	25
GPU SYSFS files.....	25
lwkgpu and lwkgpu_mask.....	26
Lwkgpu_request and lwkgpu_request_mask .....	26
Lwkgpu_reserved and lwkgpu_reserved_mask.....	26
Lwkgpu_numa .....	26
Lwkgpu_usage_count.....	26
GPU Resource Designation .....	27
GPU Resource Reservation .....	27
Build Configuration .....	28
Boot Parameters .....	28
Kernel block diagram .....	29
Run Queue .....	29
Assimilation.....	29
Idle thread.....	30
Core Sleep State.....	30

Round-Robin dispatching.....	31
Scheduling policy .....	31
YOD Scheduling policy option .....	31
Priority.....	31
Priority map .....	32
System call sched_setaffinity behavior.....	32
Light-weight Kernel implemented System Calls .....	32
Internal YOD options.....	33
lwksched-disable-setaffinity=<errno> .....	33
lwksched-enable-rr=<time_quantum> .....	33
lwksched-stats=<level> .....	33
util-threshold=<X:Y> .....	34
overcommit-behavior=<behavior> .....	34
one-cpu-per-util .....	35
idle-control=<MECHANISM,BEHAVIOR> .....	35
cmci-control=<threshold,poll>.....	35
enable-balancer=<balancer_type><parm1>, <parm2>, <parm3>.....	36
Debug Aids .....	36
Counters.....	36
PID .....	36
CPUID .....	36
THREADS .....	37
CPUS.....	37
COMPUTE_COMMIT .....	37
UTIL_COMMIT.....	37
CPU_MAX_RUNNING .....	37
GUEST_DISPATCH .....	37
TIMER_POP .....	37
SETAFFINITY .....	37
PUSHED .....	37
BALANCER_ACTIONS.....	37
Trace-point Events .....	37
mos_clone_cpu_assign .....	38

mos_timer_tick .....	38
mos_idle_init .....	38
mos_cpu_commit .....	39
mos_cpu_uncommit .....	39
mos_cpu_select_unavail.....	39
mos_cpu_select .....	40
mos_util_thread_assigned.....	41
mos_util_thread_pushed.....	41
mos_assimilate_launch.....	41
mos_clone_attr_active .....	42
mos_clone_attr_cleared .....	42
mos_assimilate_guest.....	42
mos_assimilate_idle.....	42
mos_giveback_thread.....	43
mos_select_main_thread_home .....	43
mos_mwait_cstates_configured.....	44
mos_mwait_idle_entry .....	44
mos_mwait_idle_exit.....	44
mos_halt_idle_entry .....	44
mos_halt_idle_exit.....	45
mos_poll_idle_entry .....	45
mos_poll_idle_exit.....	45
mos_balancer_tick_start .....	45
mos_balancer_tick_stop.....	45
mos_balancer_tick.....	46
mos_balance .....	46
Set Affinity Disable .....	46
Appendix .....	47
Utility Thread Application Programmer's Interface.....	47
Why use this API?.....	47
Utility Thread API Compiling and Linking.....	47
Linux compatibility.....	47
Behaviors and Locations .....	47

Behavior .....	47
Locations .....	48
Specifying a Location.....	48
Location Key Example .....	48
UTI Attribute Object.....	48
Initializing and Destroying the Attribute Object .....	49
Setting behaviors .....	49
Setting locations.....	49
Determining results.....	49
Usage Example.....	50
Interactions between pthread_attr and uti_attr .....	51

## Revision History

Date	Doc Rev	mOS version	Modified	Description
01/01/2018	0.1	0.5	Attinella	Initial version
03/30/2018	0.2	0.5	Attinella	Add machine check polling, interrupt information
04/09/2018	0.3	0.5	Attinella	CPU offline additions from Sharath. Formatting changes.
06/11/2018	0.4	0.6	Attinella	Clock source watchdog. UTI API specifics, sched_setaffinity, build configuration, boot parameter recommendations, idle thread C-state management.
08/09/2018	0.5	0.7	Attinella	Idle task control, startup. Additional ftrace directory, trace-points. Minor corrections.
12/10/2018	0.6	0.7	Attinella	rename syscall CPUs to utility CPUs
07/02/2020	0.8	0.8	Attinella	Round-robin scheduling activated by default. Execute all syscalls locally, no migration by default.
12/21/2020	0.9	0.9	Attinella	Balancer support, syscall migration changes
08/17/2021	0.9.2	0.9.2	Attinella	Remove system call migration. Grammar corrections.
05/23/2022	0.9.2	0.9.2	Attinella	Add GPU resource management

## Glossary

Term	Definition
CMCI	Correctable machine check interrupt.
commit	The assigning of a CPU home for the mOS thread. The commit will persist across blocking and sleep conditions encountered by the thread. The scheduler will be dispatching strictly on the committed CPU. The scheduler tracks the number of commits against each CPU and uses this information when choosing a CPU for a thread.
compute thread	A primary computational thread within an application
CPU	In the context of this document, a CPU refers to a hardware thread within a hyper-threaded core of a multi-core computer.
C-state	Idle power saving states, in contrast to P-states, which are execution power saving states.
GPU	Graphics processing unit typically used as an accelerator by HPC applications
HPET	High Precision Event Timer. One central timer exists in a system. Multiple compare registers used by individual CPUs to provide an interrupt. Accessed through MMIO. Slower access than TSC.
KNL	2 <sup>nd</sup> generation Intel Xeon Phi Processor, code-name: Knights Landing
Linux CPU	A CPU that is not configured as a light-weight kernel CPU. This can also include CPUs configured as system call targets for the light-weight kernel CPUs.
LWK	A light-weight kernel.
LWKCPU	A CPU owned/controlled by a light-weight kernel.
lwkctl	The LWK partition configuration tool. Used to configure resources for an LWK partition. See man page for more info.
node	One computer within a cluster of computers.
NUMA node NUMA domain	A portion of memory that exhibits common throughput and latency characteristics relative to the CPUs within the computer. A CPU located close to a NUMA node may be able to access its memory faster than a CPU located far from the NUMA node.
OneAPI Level Zero	The set of APIs used to access/use the GPU accelerators. Level Zero represents the lowest level user interfaces to the accelerators. Run-times are the primary user of the OneAPI interfaces.
over-commitment	When more than one thread is committed to execute on an LWKCPU.
P-states	The P-states are voltage-frequency pairs that set the execution-time speed and power consumption of the processor.
tile	A physical grouping of core(s) and cache. For example, on KNL, two cores compose a tile, and there is an L2 cache on the tile shared by the two cores.
TSC	Time Stamp Counter. A 64 byte timer that exists in each CPU. At boot, the TSC is synchronized across all of the CPUs. Fast read/write via machine instructions.
utility CPU	A Linux CPU that has been configured as a target CPU for executing system calls that originated on LWK CPUs and to be used to host utility threads.
utility thread	A helper thread created by a runtime or an application, not considered one of the primary computational threads within an application.
yod	The process launcher in the mOS environment. See man page for more info.



## Introduction

The primary goal of the scheduler is to provide a low noise environment that can efficiently and intelligently place, dispatch, and run the processes of an HPC application. HPC applications typically contain synchronization points that require many if not all processes across all nodes to reach the same location before proceeding to the next phase. Because of this synchronization, it is very important for the operating environment to not introduce random noise or delay any of the threads during its parallel computational phases since delaying one thread on one CPU could potentially delay the progress of thousands of processes across hundreds of nodes, wasting valuable CPU resources while they wait for a lagging process to reach the synchronization point.

## mOS Scheduler Attributes

### Low noise, predictive scheduling

In the mOS scheduler, no repeating timer ticks are set, and all configurable interrupts are redirected to Linux CPUs. No Linux kernel threads and no user processes other than the application launched through the YOD job launcher will run on the light-weight kernel CPUs.

### Fast function adaptation

The scheduler code is isolated from the Linux code base in a way that promotes fast code modifications allowing mOS to provide new scheduler functionality in a timely manner. Many low-level building blocks of the Linux kernel are leveraged, avoiding the reinventing of functions that do not need to be differentiated from Linux.

### Behaviors optimized for HPC and AI

HPC performance gains are realized by optimizing the scheduler design for environments where there is no over commitment of CPU resources by processes. Linux kernel strives to ensure available CPU resources are equally and fairly utilized across many processes which often is at the expense of individual thread performance. This involves frequently interrupting threads and attempting to balance the overall workload by pushing or pulling threads to CPUs.

AI performance gains are realized by managing overcommitted CPUs at a process granularity. This can be accomplished in mOS due to the underlying design of a per-process reservation of CPU resources. In addition, preferential treatment is given to CPU-intensive threads. These threads are not penalized as would typically occur in the Linux scheduler.

### Topology aware thread placement

The mOS scheduler understands the relationship between CPUs, NUMA domains, tiles, and cache levels. The scheduler will assign threads to CPUs following the rules provided at launch time. Various packing and spreading layout options are provided. Unlike a run-time that can only control the placement of its own threads, mOS can manage the threads created within the entire application, crossing run-time boundaries. This system view avoids performance problems introduced when one runtime is placing threads without regards to other thread placement actions occurring outside of its limited scope.

### Flexible and simple handling of utility threads

Handling of the extra *utility* threads that certain applications and runtimes create as helpers has often been problematic in the HPC world. The mOS scheduler provides two different approaches to deal with

these types of threads. The threads can be heuristically identified as utility threads or can be explicitly identified by the application and runtime using the utility thread APIs. This is a set of APIs being jointly developed across multiple light-weight kernel projects. The utility thread APIs allow the caller to intelligently place the utility threads and provide hints as to its runtime behavior without requiring the user to understand the detailed topology of the underlying hardware.

### Strict CPU affinity

The scheduler will make every effort possible to keep a thread executing alone on a specific CPU. If it is possible to eliminate over-commitment on a specific CPU, a thread will be moved to an un-committed CPU at a wakeup point. The scheduler considers a CPU as committed even if a thread that was previously running on that CPU has blocked and is not currently on the run queue of that CPU. We want that blocked thread to wake up and find that same CPU available.

### Support of pthread libraries and runtimes

The mOS scheduler provides full support of the pthread libraries and associated affinity operations. If the caller requests to be placed on a specific CPU, the request will be honored. Also setting thread priorities and a round robin scheduling policy is supported.

### Inter-operability with the Linux scheduler

The threads running in the light-weight kernel's scheduler can interact with threads running in the Linux kernel's scheduler. This includes blocking, wait lists, and waking. Inter-operability is accomplished by sharing many of the scheduler data structures and low-level software mechanisms.

### Compatible with many existing Linux debug/performance tools

Many of the existing Linux performance tools and debug tools will operate on processes running in the light-weight kernel, including the collection of various performance counters.

### Dynamically configure CPU resources

The mOS kernel allows the setting of the CPUs that will be controlled by the light-weight kernel and the CPUs that will be controlled by Linux, including the Linux CPUs that will be acting as the target for the migrated system calls. There are kernel boot-time parameters that can be specified to create an initial CPU configuration. Also, the configuration can be created and modified after the boot has been completed by using configuration commands. Similar dynamic configuration capabilities exist for memory.

### Internal options for experimentation

The scheduler currently supports several internal options that can be specified when the job is launched through YOD. The YOD job launcher implements a flexible approach that allows new options to be easily passed through to the scheduler. This is useful when experimenting and analyzing behavior. With minimal code changes in the scheduler a new option can be supported through the user's job launch interface.

## Design

### CPU Assignment

#### Node Level

At the node level, CPUs to be designated as light-weight kernel CPUs can be specified in two ways:

1. *lwkcpus* kernel boot parameter

## 2. *lwkctl* -c command

### *lwkcpus kernel boot parameter*

This parameter describes which CPUs are to be light-weight kernel CPUs and which CPUs are to be utility CPUs. An example specification for a 68 core KNL node would be:

```
lwkcpus=1.52-67,256-271:69.120-135,188-203:137.2-17,206-221:205.70-85,138-153:19.20-35,224-239:87.88-103,156-171:155.36-51,240-255:223.104-119,172-187
```

In this example, CPUs 1, 69, 137, 221, 205, 87, 155, and 223 will be utility CPUs running in the Linux scheduler and CPUs 52-67, 256-271, 120-135, 188-203, 2-17, 206-221, 70-85, 138-153m 20-35m 224-239m 88-103m 156-171, 36-51, 240-255, 104-119, and 172-187 will be light-weight kernel CPUs. The CPUs 0, 68, 136, 18, 86, 154, 204, and 222 will be exclusively for Linux use.

### *lwkctl command*

The *lwkctl* command allows the CPU configuration to be modified/created after the operating system is booted. This is the preferred and most flexible approach to creating a light-weight kernel partition. The command supports an 'lwkcpus=auto' option for the CPU specification. When 'lwkcpus=auto' is used for the specification, mOS will determine the CPU specification based on the topology of the host. When not using the 'lwkcpus=auto' option, the syntax of the CPU specification is the same as the kernel boot parameter. In this following example, a light-weight kernel partition is created with the same CPU specification shown in the previous kernel boot parameter example:

```
sudo lwkctl -c 'lwkcpus=1.52-67,256-271:69.120-135,188-203:137.2-17,206-221:205.70-85,138-153:19.20-35,224-239:87.88-103,156-171:155.36-51,240-255:223.104-119,172-187  
lwkmem=0:16G,1:16G,2:16G,3:16G,4:3968M,5:3968M,6:3968M,7:3968M'
```

To show the existing partition, the following command is used:

```
lwkctl -s
```

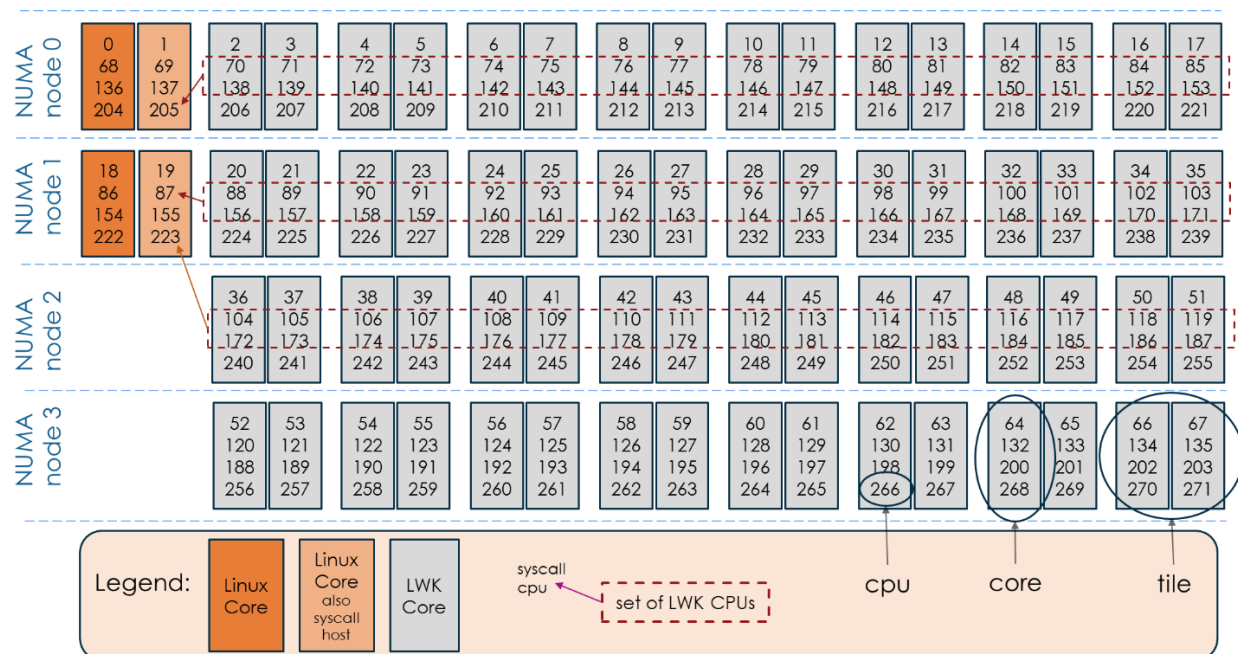
The result of the show request:

```
mOS version : 0.6  
Linux CPU(s): 0-1,18-19,68-69,86-87,136-137,154-155,204-205,222-223  
LWK CPU(s): 2-17,20-67,70-85,88-135,138-153,156-203,206-221,224-271  
Utility CPU(s): 1,19,69,87,137,155,205,223  
LWK/Linux GPU(s): 0-5 [ 6 GPU(s) 2 TILES/GPU ]  
LWK Memory (KB): 16777216 16777216 16777216 16777216 4063232 4063232 4063232 4063232
```

See the man page for *lwkctl* for additional information.

### Node CPU configuration diagram

The following diagram depicts the example system configuration specified in the previous sections:

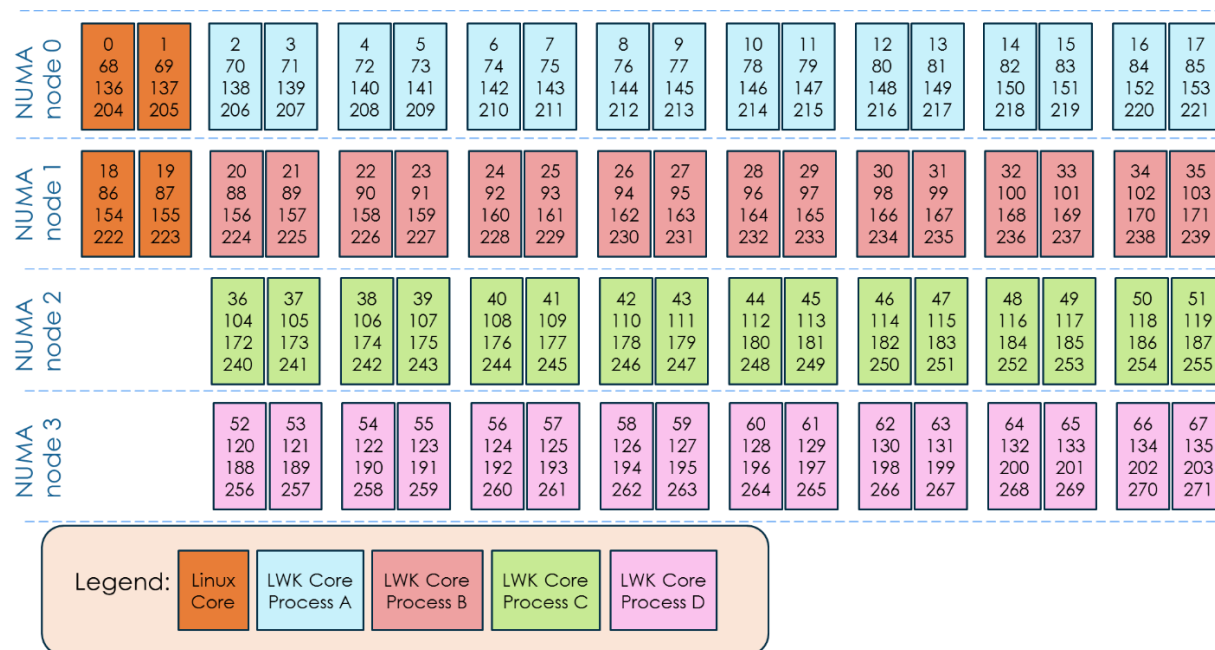


### Process Level

At the process level, the mOS job launcher, YOD, manages which CPUs are reserved for each created process. For example, on a 68 core KNL booted in SNC4 flat mode, if YOD is launched 4 times, with each request asking for 1/4<sup>th</sup> of the system's resources, and if each of the 4 domains has an equal number of light-weight kernel CPUs, then each process launched by YOD will reserve CPUs contained within one domain. After the process begins running, the mOS scheduler will manage how the threads within the process are assigned to CPUs.

### Process CPU reservation diagram

The following diagram depicts the process allocation example described in the previous paragraph.



### Potential Conflicts

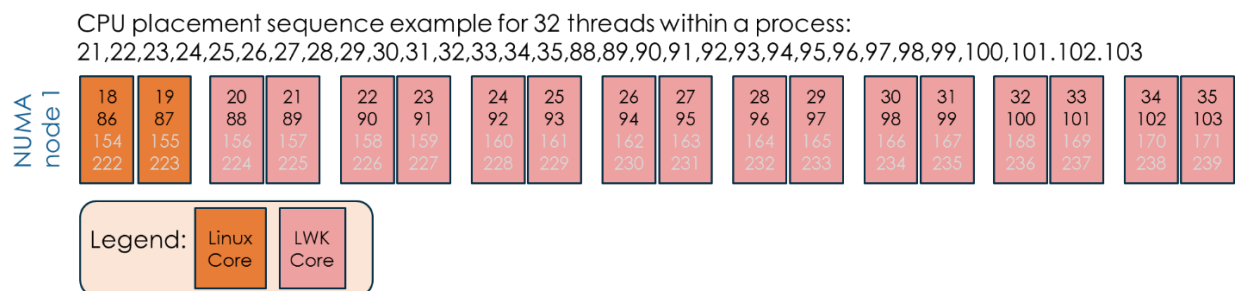
If YOD is launched through a higher-level job launcher, there may be conflicts that need to be avoided. For example, when using the *mpirun* command within the Intel MPI package, the `_MPI_PIN=off` environment variable must be set to prevent the Intel MPI runtime from attempting to set the CPU affinity for ranks within the MPI job being launched. This action is in direct conflict with what YOD is doing. Also, Intel MPI has no knowledge of which CPUs are light-weight kernel CPUs and which CPUs are Linux CPUs therefore it cannot make the correct selections when running under mOS.

### Thread Level

Thread level decisions on what CPU to use are made by the scheduler, along with help from YOD. When YOD launches a process, it sends down to the kernel a sequence list of the CPUs that were reserved for the process. The order of this sequence list is based on the thread layout order that was decided at launch time. By default, the layout is a scatter layout. With this layout, threads are spread in order of NUMA domains, tiles, cores, and CPUs, across the available CPU resources for the process. The YOD job launcher allows for the specification of various layout options. The layout options are similar in capability to the Intel OpenMP `KMP_AFFINITY` layout options, however they are managed system-wide and not limited to the threads placed by just the OpenMP run-time. See the YOD man page for more details on the various layout options. As the process creates pthreads, the scheduler will assign those pthreads to the CPUs using the sequence list as the search order. It will always search for the least committed CPU using this search order. Since runtimes and applications are free to use `setaffinity` operations, the scheduler cannot assume that the next CPU in the sequence list is available. It re-searches the sequence list starting at the beginning for the least committed CPU. When the scheduler selects a CPU, the new thread will be started directly on that CPU.

### Thread placement example

The following diagram depicts a thread placement sequence in which the process contains a total of 32 threads. In this example, the YOD layout option ‘-layout=core,tile,node,cpu’ was specified. This indicates that the layout will first populate each core within a tile. When a tile fills, it will move to the next tile within the same NUMA domain. After all the tiles have been filled in that NUMA domain, it will move to the next NUMA domain. In this example the process configuration only contains CPUs from one NUMA domain. After all cores have been assigned a thread, the scheduler will begin assigning additional CPUs (hardware threads) within each core. This is the resulting sequence:



### Utility Threads

Utility threads are helper threads that run-times and applications sometimes create to assist in the executing of the application. Some examples are MPI progress threads, OMP monitor threads (now obsolete), and PSM2 RCVTHREADs. When running an HPC application, we typically want to keep the threads running the parallel computation work resident on a CPU and we do not want them interrupted for any reason, including the running of a utility thread. Ideally, the utility thread would run on some other CPU, and it would run in an environment that is a good match for the behavior attributes and memory access patterns of that utility thread. To satisfy this need, the Utility Thread API was created.

### Utility Thread API

The definition of the UTI is an ongoing group effort across multiple light-weight kernel platforms. The advantages of using the Utility Thread APIs are the following:

1. Keeps these extra threads from interfering with computational threads.
2. Allow grouping of utility threads across the ranks in a node.
3. Callers do not require detailed system topology knowledge or O/S scheduler knowledge to provide desired placement and scheduling.
4. Callers do not require awareness of other run-time or application thread placement actions to avoid conflicts.

The API is structured like the pthread\_create API and can easily be adapted by run-time and application code that currently create their utility thread using pthread\_create. The API will let you specify a behavior attribute for the utility thread. This will cause the scheduler to optimize its scheduling behaviors for the thread. The currently supported behaviors:

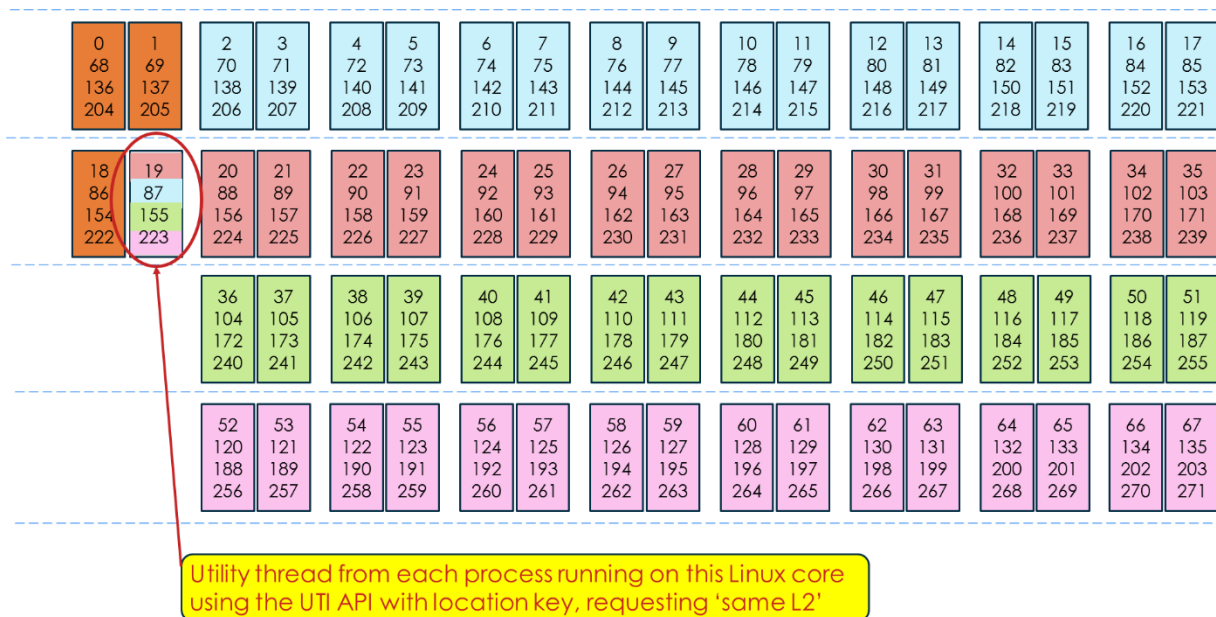
1. CPU intensive, e.g., constant polling.
2. Expects high scheduling priority.
3. Expects low scheduling priority.

4. Does not play nice with other threads, infrequent yields, and blocks.
5. Expects to run on a dedicated CPU.

The API also allows the caller to suggest a preferred location, without necessarily requiring an explicit location specification. The supported locations are:

1. Place on same L1, L2, L3, NUMA domain using a shared location key.
2. Place on the same or different L1, L2, L3, NUMA domain as creator.
3. Place on a specific NUMA domain.
4. Place on a Linux CPU.
5. Place on a light-weight kernel CPU.
6. Place on a CPU that is handling fabric interrupts.

The following diagram shows an example of using the shared location key, “Same L2”, and “Place on Linux CPU”. This will place the utility threads from each process (rank) on Linux CPUs that share the same L2 cache.



See the Appendix for more information regarding the usage of the UTI API.

#### Heuristic utility thread placement

Another supported mechanism to place utility threads is heuristic placement. This was the initial method used to support utility threads prior to the introduction of the UTI API. It is still useful in environments where the users of utility threads have not adopted the UTI API (as of this time, no applications or runtimes have adopted these APIs). With this method, the number of utility threads that the application will be creating is specified to YOD using the ‘-u <n>’ keyword. The heuristic used is simply that the scheduler will assume that the first <n> pthreads created within the process will be the utility threads. This simple heuristic can only be used in limited situations. By default, the threads will be assigned to CPUs by walking the CPU sequence list from the end towards the beginning. This will keep the layout assignment of the worker threads maintained. If a worker thread is later created and the only remaining location to place

the worker thread is a CPU that contains a utility thread, that utility thread will be pushed to a Linux CPU and the worker thread will be placed on the light-weight kernel CPU.

There is an internal scheduler option that can provide more advanced placement of heuristically identified utility threads. This option can be used to study the consequences of utility thread placement. The option is specified through YOD in the following manner: “-o util-threshold=<X:Y>”. The X value indicates the maximum number of light-weight kernel CPUs that can be used for hosting utility threads. The Y value represents the maximum number of utility threads allowed to be assigned to any one light-weight kernel CPU. Some examples: A value of “0:0” will prevent any utility threads from being placed on an LWK CPU and force all utility threads to be placed on the Linux CPUs that are defined to be the utility CPUs. A value of “-1:1” will allow any number of LWK CPUs to hold utility threads, however only a maximum of one utility thread will be assigned to any LWK CPU.

#### *Running on a Linux CPU*

When a utility thread is placed on a Utility CPU, it will be given to the Linux scheduler for scheduling. This is true for both UTI API placement and heuristic placement actions. The thread will be changed from the mOS scheduling policy/class to the ‘Fair’ scheduling policy/class and by default given a nice value of (-10). This will allow it to be fairly scheduled with other utility threads and allow it to have preferred scheduling treatment over most other Linux processes (with a nice = 0) that may be running on that CPU. If the caller has requested high scheduling priority, the nice value of the utility thread will be set to (-19). If the caller requested low scheduling policy, the nice value of the utility thread will be set to (+20). Once a utility thread is placed on a Linux CPU, it cannot return to the mOS scheduler. It will live its remaining life in the Linux scheduler running on Linux CPUs.

By default, the CPUs allowed mask within the utility thread is set to include all the available utility CPUs that satisfy the location requested. For example, if the caller uses the UTI API and specifies that the utility thread should be placed in NUMA domain 0, then the utility thread will be set to allow execution on all the utility CPUs that are in NUMA domain 0. The specific CPU to be used for execution of the utility thread is then decided by the Linux Fair scheduler. This behavior can be overridden using the following YOD option: “-o one-cpu-per-util”. When this option is used, the scheduler will choose the least committed utility CPU from the list of CPUs that satisfy the location request. The CPUs allowed mask within the utility thread will be set to only include that one CPU. The Linux Fair schedule will then be restricted to scheduling the utility thread on only that CPU.

#### *Over-commitment*

Over-commitment is a situation in which more than one thread is assigned to run on a CPU. This can be caused by a variety of reasons:

1. The application/runtime has created more pthreads than the number of available CPUs reserved for the process/rank.
2. The application/runtime has used a mechanism such as pthread\_setaffinity to explicitly place threads on specific CPUs.
3. The application/runtime has used the UTI API to direct the placement of utility threads in such a way that over-commitment could not be avoided.

The mOS scheduler has default behaviors to deal with over-commitment situations. Internally the mOS scheduler tracks the commits of compute threads and utility threads separately. This provides flexibility in dealing with over-commitment.

#### More compute threads than CPUs

As compute threads are created, they are assigned to CPUs according to the sequence list provided by YOD (as previously described). If more threads are created after all the CPUs have been assigned to a thread, the sequence list will be followed again, starting with the first CPU in the sequence. Note that the first CPU in the sequence list is the CPU initially assigned to host the main thread of the process.

#### Explicit compute thread placement

If compute threads are explicitly assigned to CPUs using `setaffinity` operations (typically through `pthread_setaffinity`), then regardless of the current commit level, the placement will be honored. If a thread was placed somewhere in the middle of the sequence list and then compute threads are created and assigned a CPU by the mOS scheduler, it will skip over the CPUs containing the explicitly placed threads if there is a thread with a lower commitment level later in the sequence list. If a thread was explicitly placed on a CPU that was already assigned to a thread by the mOS scheduler, then the CPU will be overcommitted for at least a period, up until the thread encounters a wakeup condition. At that time the mOS scheduler will perform *wake-up balancing*.

#### Wake-up balancing

When a thread is being awakened from a block condition, the mOS scheduler will look to see if its CPU home is also occupied by another thread. If it is, then a search will be done to see if there are any other un-committed CPUs within the process. If an un-committed CPU is found, this CPU will become the new home for this thread, eliminating the over-commitment condition. This wake-up balancing is in place to address the conditions when the run-time is explicitly placing some threads, and mOS is placing other threads without awareness of the runtime's intentions. The wake-up balancing also will assist in the balancing of threads across the available CPUs for rare situations in which the mOS scheduler did not initially make the best CPU selections due to threads being created concurrently with a process.

#### Thread placement through Utility Thread API

When the Utility Thread API is used to place threads, the requested location will be honored even if it results in over-commitment. For example, if "Same L2" is requested, and all the CPUs located under that L2 are committed to a thread, the location request will be satisfied, and a CPU will be over-committed.

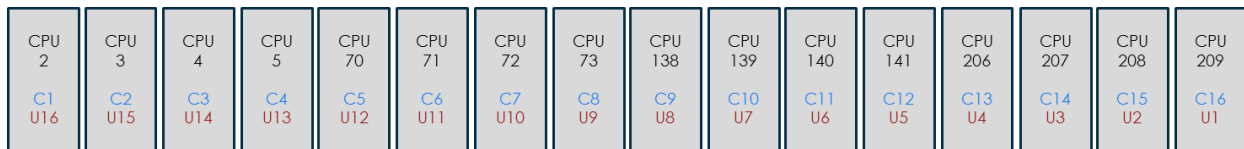
#### Compute and Utility thread interactions

When a mixture of compute threads and utility threads exist within a process and all the available CPUs are committed (but not yet over-committed), the following behaviors are true:

- When creating an additional compute thread:
  1. If a utility thread was previously created without any explicit location information, it will be on a push list. The mOS scheduler will see if there are any utility threads that can be pushed to a utility CPU. Pushing will be done one by one until a CPU becomes uncommitted or until there are no other utility threads available to be pushed.

2. If no CPUs are un-committed after the push attempt, the mOS scheduler will try to find a CPU that does not have a compute thread assigned to it (i.e., it may have a utility thread assigned). The desire is to avoid placing two compute threads on the same CPU. If none are available, the least committed CPU will be selected.
- When creating an additional utility thread:
    1. If no location request is provided, or if the location request is for a Linux CPU, the utility thread will be placed on the Linux CPU that has the least number of utility thread commits.
    2. If a location request results in an LWKCPU location, the scheduler will search the CPU sequence list from the end towards the beginning to find the CPU with the least number of utility thread commits which satisfies the location request.

The following diagram shows how the compute threads and utility threads will be sharing the CPUs. This assumes the utility threads were created with the LWK CPUs identified as the target location (not Linux CPUS). Regardless of the order of when the threads are created, compute threads first, utility threads first, or alternating, the default mOS scheduler behavior will produce the following layout. The CPU IDs in the following example are based on a 2 core process created on a 68 core KNL.



## Thread Push/Pull Balancers

Many of the newer AI workloads do not fit the mold of a traditional HPC application. These new applications, many of them python based, often create many more threads than the number of available CPUs. It is also difficult to statically assign these threads as either compute or utility threads as they perform various tasks throughout the life of the application. When executing these applications on mOS, enabling one of the available balancers may improve performance.

### Pull Balancer

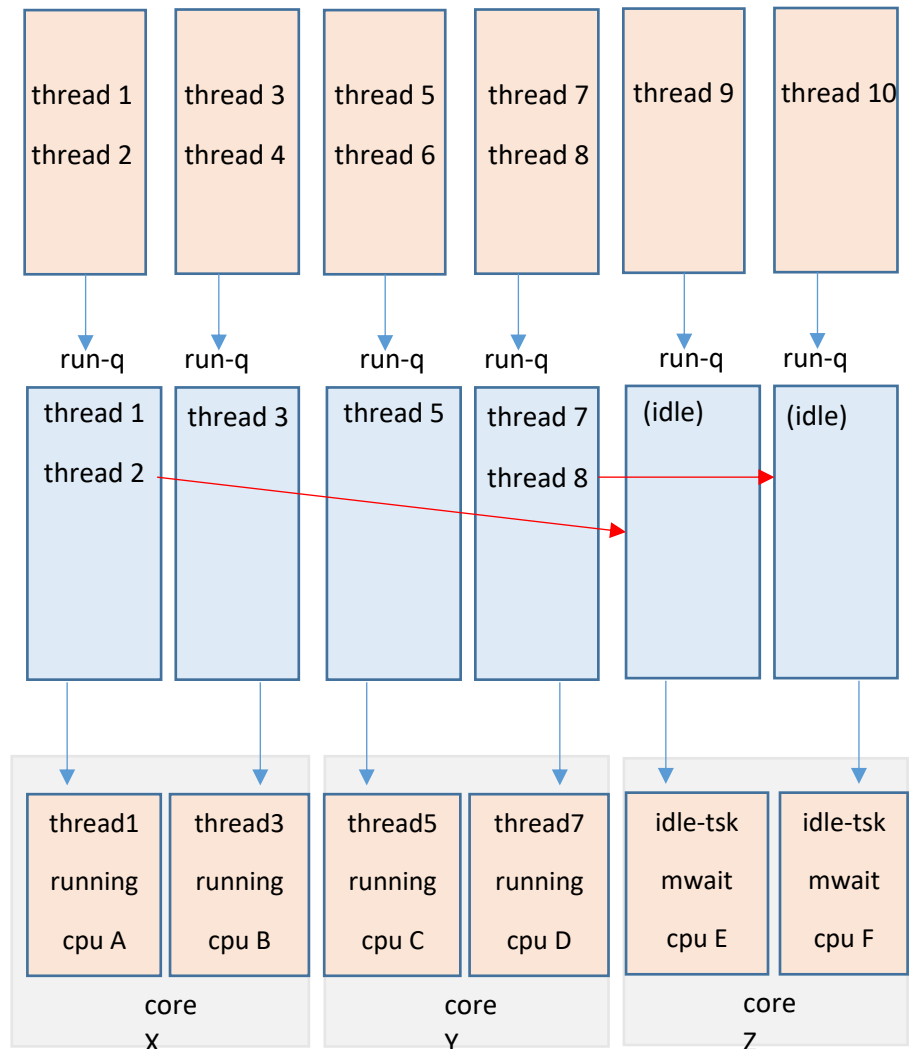
In the Pull balancer, balancing actions occur whenever a CPU enters idle and periodically when in idle for an extended time. The pull balancer is enabled when the process is launched using YOD. There are three parameters that can be specified when enabling the pull balancer:

Parameter 1: This is the timer-tick spacing for when a CPU is sitting in the idle state for an extended period. While the CPU is in the idle state, a timer interrupt will occur every “parameter 1” milliseconds. At this time, the other CPUs reserved by this process will be inspected to see if work should be pulled from them. The default value for this parameter is 100 milliseconds.

Parameter 2: A pull of a thread from a CPU will not occur unless that CPU has been overcommitted with more than one running thread for a sustained period of “parameter 2” milliseconds. The default value for this parameter is 20 milliseconds.

Parameter 3: The number of microseconds that the balancer will wait upon entry to the idle state before it attempts to pull threads to the idle CPU. The default value for this parameter is 0 microseconds.

1. CPU E entering idle looks for an overloaded CPU
2. CPU A load is determined to exceed thresholds and is the highest, so a non-running thread is pulled to CPU C
3. CPU F timer popped while in idle, looks for an overloaded CPU.
4. CPU D load is determined to exceed thresholds and be highest, so the non-running thread is pulled. to CPU F



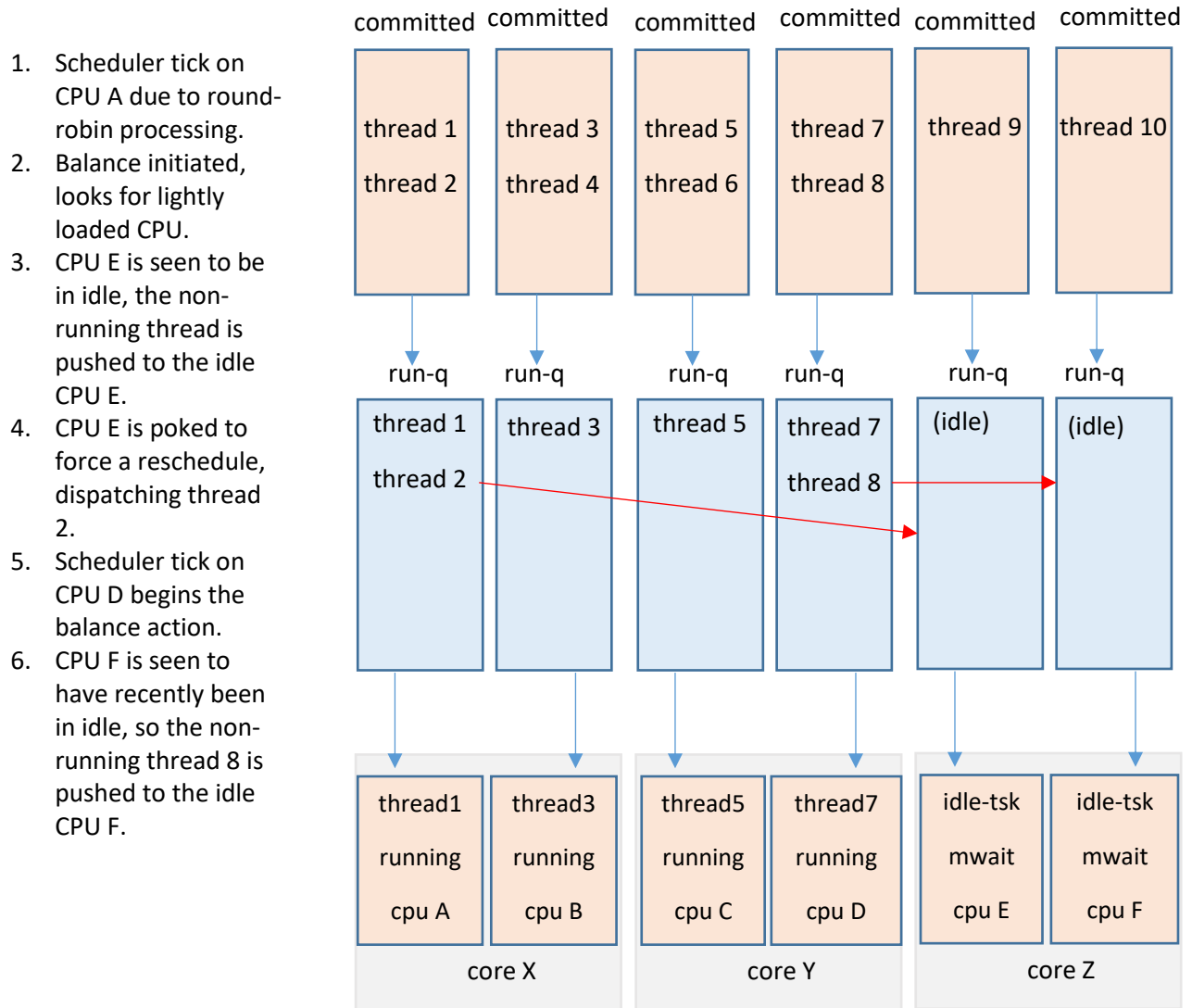
### Push Balancer

In the push balancer, balancing actions occur when the round-robin timer tick occurs in a CPU that is overcommitted. At that time, the other CPUs associated with the current process will be inspected to see if a lightly utilized CPU exceeding the specified threshold exists. If so, a non-running thread on the overcommitted CPU's run queue will be pushed to this lightly utilized CPU. There are three parameters that can be specified when enabling the push balancer:

Parameter 1: Consider pushing only if a target CPU load can be found that is less than the  $(\text{current\_cpu\_load}) / (2 * \text{parameter\_1})$  where  $\text{cpu\_load} = (\text{number\_running\_threads}) * \text{time\_since\_idle}$ . The default value for this parameter is 12.

Parameter 2: Consider pushing only if the current CPU has been overcommitted with running threads for a sustained period of "parameter 2" milliseconds. The default value for this parameter is 20 milliseconds.

Parameter 3: The number of milliseconds needing to expire before a CPU that received a pushed thread is eligible to accept another pushed thread. The default value for this parameter is 10 milliseconds.



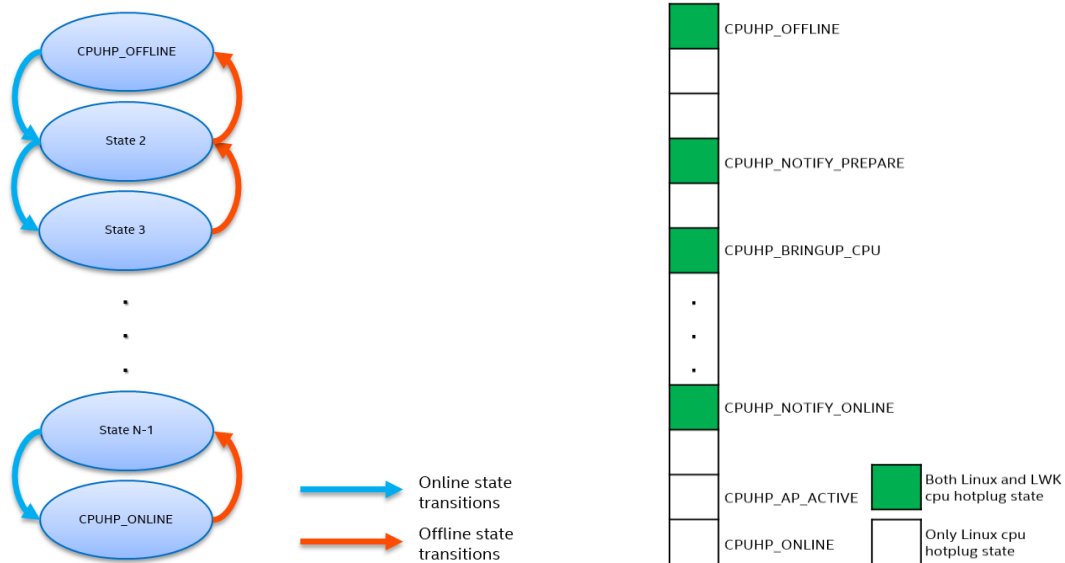
## CPU Isolation and configuration

CPU isolation between the light-weight kernel and the Linux kernel is accomplished using the following mechanisms:

1. Linux CPU hot-plug
2. Linux modifications and mOS Scheduler code
3. Dynamic migration of interrupts

## Linux CPU hot-plug

The Linux kernel supports the logical off-lining and on-lining of CPUs through CPU hot-plug mechanism. Off-lining or on-lining a CPU involves transition of the CPU through a series of CPU hot-plug states wherein each state can be considered as a step during the transition as shown below.



Every hot-plug state is registered by any one of the kernel components either statically during the kernel build or dynamically during the kernel boot. These steps can be further classified as critical and non-critical steps. Critical steps are those which need to be executed to have a functional CPU with primitive kernel services ex: states that initialize hardware, states registered by scheduler core. Non-critical steps are optional states ex: those that are being registered to parallelize work in the kernel such as bounded/unbounded work queues.

The mOS takes advantage of this capability to keep the Linux user processes and kernel threads (except for few critical kernel threads which are allowed by mOS scheduler) off the CPUs designated as light-weight kernel CPUs. To configure an on-lined Linux CPU as an LWK CPU, first it will be off-lined through all the Linux hot-plug states. This migrates all the user and kernel threads away from that CPU which were previously scheduled or running on that CPU. Subsequently the CPU will be brought up again from the off-lined state but this time transitioning only through a critical set of steps (CPU hot-plug states) that are necessary for it to be functional to run LWK processes. This results in a low noise and isolated CPU environment for LWK. When returning the CPU to Linux it will be brought down through the critical set of states and again brought up through all Linux hot-plug states to be a fully functional Linux CPU.

This mechanism allows the movement of CPUs to and from the light-weight kernel without requiring a reboot of the OS. No jobs can be executing on the LWK CPUs during CPU configuration actions. Once the CPUs are being brought up as LWK CPUs they are no longer available to Linux scheduler to schedule either user tasks or kernel threads and cannot be hot-plugged (off-lined/on-lined) through standard Linux SYSFS interfaces until these CPUs are handed back to Linux by deleting LWK partition.

## Linux modifications and mOS Scheduler code

To maintain run-time isolation and maintain interoperability with Linux, the mOS scheduler class was implemented as a scheduler plugin in an analogous manner as the other scheduler classes: fair, runtime,

and deadline. However, this level of integration was not enough to provide the desired level of scheduler isolation from the Linux scheduler. Additional isolation is achieved through modifications to Linux kernel code. The following table shows the Linux parts that have been modified along with the number of modified lines.

#### *Modified Linux Source Files*

<b>Part</b>	<b>Lines of code added/modified</b>
arch/x86/kernel/cpu/mce/intel.c	44
arch/x86/kernel/cpu/mce/core.c	54
arch/x86/include/asm/syscall_wrapper.h	44
include/asm-generic/vmlinux.lds.h	1
include/linux/sched.h	50
include/linux/syscalls.h	1
include/linux/workqueue.h	6
kernel/fork.c	8
kernel/sched/core.c	203
kernel/sched/sched.h	94
kernel/time/clocksource.c	41
kernel/workqueue.c	13
kernel/sched/Makefile	1

#### *New mOS Scheduler Source Files*

<b>Part</b>	<b>Lines of code</b>
kernel/sched/mos.c	3685
kernel/sched/mos.h	286
mOS/Kconfig	26
include/trace/event/lwksched.h	683
include/trace/event/lwkwake.h	169
include/linux/mos.h	73
include/linux/mosras.h	18

#### *Correctable Machine Checks*

The default behavior in the Linux kernel is to have correctable machine check interrupts (CMCI) enabled in each CPU with a threshold of one in each bank and to have a polling interrupt generated every 5 minutes in each core. The polling function performs a check of the machine check banks that do not support CMCI within that core, possibly resulting in logging the machine check events. When an application is executing on an LWK CPU, we do not want this potential noise. Therefore, when an LWK process is launched, the threshold for CMCI's will be changed from 1 to the maximum allowed by the hardware platform for the LWK CPUs reserved for that process. Also, the polling timer interrupt will be disabled on these CPUs. When the LWK process exits, CMCI interrupt threshold will be restored, and the polling timer will be re-enabled. Also, on process exit an immediate check of the machine check banks will be performed to flush out any correctable event data that may have been recorded in the machine check banks during the execution of the application. Depending on which CPUs are LWK CPUs, there may be CPUs that will immediately handle

correctable machine check conditions occurring during the running of a mOS process. Any machine check banks that are shared across CPUs, for example DDR banks, will typically be managed by the lowest CPUID that can access that DDR bank. For example, on a 2-socket system, the first CPUID in each socket would own the CMCi interrupt for the DDR that is controlled by that socket. If that CPU happens to be configured as a Linux CPU (not an LWK CPU) and if an error occurs during the running of an mOS job, the error reporting and logging will not be delayed until the end of the job. Instead, the Linux CPU will handle the condition immediately.

The scheduler supports YOD option, '-o cmci-control=<threshold, poll>', that can be specified to modify the previously describe behavior.

threshold:

- The number of correctable machine check events that must occur in a machine check bank before a correctable machine check interrupt is delivered to a LWK CPU that is hosting an mOS process. The value specified must be less than 32768. A value of 0 disables interrupt delivery to all LWK CPUs hosting the mOS process.
- 'disable-cmci': same as threshold = 0
- 'max-threshold': Set threshold to the maximum allowed value. This will typically be 32767 on newer hardware platforms. (note this is the new default behavior)

poll:

- 'disable-poll': Polling will be disabled on the LWK CPUs that are hosting the mOS process.
- 'enable-poll': Polling will be enabled and operate with the same interval, behavior, and control as in the Linux OS

### [Clock source watchdog](#)

By default, the Linux kernel runs a clock source watchdog. This watchdog executes within a timer interrupt handler every 0.5 seconds. At the end of its execution, it sets its timer on the next online CPU in the online CPU mask. This timer interrupt continues to fire sequentially on each online CPU continuously, wrapping on the CPU online mask. The watchdog reads the time from the currently designated clock source from the current CPU's Time Stamp Counter (TSC). It generates a delta from value obtained on the previously interrupted CPU, and then compares this delta to the delta generated from using the High Precision Event Timer (HPET). If the two deltas exceed a threshold of 0.0625 seconds, then the TSC is marked as unstable and all future operations to generate a time-of-day value will use the HPET, which has a slower access time than the TSC. This interrupt behavior will persist until the system is rebooted.

For mOS, the Linux kernel was modified such that the LWK CPUs do not participate in the checking of the TSC reliability. If the TSC is found to be un-reliable during the checking of Linux CPUs, both Linux and the mOS LWK will be switched to use the HPET timer. In the hardware platforms using the mOS kernel, the TSC is expected to be stable and synchronized barring a hardware failure.

## GPU Resource Management

When the LWK partition is created, all GPU devices will be designated as available management by the LWK. When a process is launched in the LWK, the mOS kernel will intelligently select and isolate a combination of GPU, CPU, and system memory resources for the process. The mOS interfaces will allow the caller to override any or all automatic resource selections performed by the mOS kernel. For example, if a specific set of GPUs have been requested by the caller, the mOS kernel will intelligently choose corresponding CPU and memory resources for that process.

Unlike CPU and system memory resources, GPU devices are not required by the Operating System (OS) for operation. Providing strong isolation of CPU and memory resources between the Linux kernel and the Light-Weight kernel (LWK) is needed to prevent the Linux kernel from using resources that the LWK kernel is giving to the HPC applications. Of lesser concern is the isolation of GPU devices between the Linux kernel and the LWK because the Linux OS will not be using GPU devices for computation. The Linux kernel will continue to have full access to the GPUs which simplifies the support of GPU tools and reduces the modifications to Linux that would be necessary for strict isolation. A typical HPC application will contain multiple ranks on a node. Each rank will likely be executing a GPU-enabled program. Isolation and balancing of resources between these ranks is important. The mOS kernel will carefully select CPU, memory and GPU resources for these ranks and will provide isolation and balancing of GPU, CPU, and memory resources between the ranks.

## GPU User Interfaces

The following sections will describe in detail how a user can control GPU management in mOS using YOD options and environment variables.

### GPU Affinity Control

In the Level Zero OneAPI interface, a process environment variable, `ZE_AFFINITY_MASK`, can be used to limit what GPU devices are visible to a specific process. This support is implemented by the LevelZero drivers and the i915 kernel mode driver. This support is functional in both Linux and LWK partitions without any mOS-specific driver modifications. The mOS kernel leverages this environment variable for use in the LWK partition in two ways. The mOS kernel will either set an appropriate `ZE_AFFINITY_MASK` into the process based on the other resource options specified, or it will select appropriate core and memory resources based on the caller supplied `ZE_AFFINITY_MASK`. These two approaches provide user flexibility and simplicity to allow an mOS-driven intelligent selection of process resources starting from either a grouping of cores or a user specified selection of specific GPU devices.

### GPU SYSFS files

The mOS kernel contains various sysfs files under the directory `/sys/kernel/mOS/`. The files are used to display information regarding the system resources available to the mOS kernel. The files are also used by the `lwktcl` and `yod` commands to generate requests to the mOS kernel for designation and reservation of system resources.

lwkprocesses  
lwkcpus  
utility\_cpus  
version  
utility\_cpus\_mask

lwkcpus\_mask  
ras/config  
ras/inject  
ras/jobid  
ras/location  
lwk\_options  
lwkcpus\_sequence  
lwkcpus\_reserved\_mask  
lwkmem\_reserved  
lwk\_util\_threads  
lwk\_config  
lwkcpus\_request  
lwkmem\_mempolicy\_info  
lwkmem\_request  
lwkmem  
lwkcpus\_reserved  
lwkcpus\_request\_mask

With the addition of the PVC GPU support, the following new files have been added:

lwkgpu  
lwkgpu\_mask  
lwkgpu\_request  
lwkgpu\_request\_mask  
lwkgpu\_reserved  
lwkgpu\_reserved\_mask  
lwkgpu\_numa  
lwkgpu\_usage\_count

#### *lwkgpu and lwkgpu\_mask*

These files show the GPU devices and sub-devices (tiles) available for reservation in the lightweight kernel (LWK) partition.

#### *Lwkgpu\_request and lwkgpu\_request\_mask*

These are write-only files used by *yod* to request a reservation for a GPU.

#### *Lwkgpu\_reserved and lwkgpu\_reserved\_mask*

These are read-only file to show GPUs reserved by at least one LWK process.

#### *Lwkgpu\_numa*

This file shows the NUMA node ids associated with the GPU devices listed in file *lwkgpu*.

#### *Lwkgpu\_usage\_count*

This file shows the number of LWK processes currently reserved to use each of the GPU devices. The counts are associated with the GPU devices listed in the file *lwkgpu*.

## GPU Resource Designation

The *lwctl -c* command is used to designate resources to the LWK partition. The designation of the GPU resources has been added to the existing designation of core and memory resources. The *lwctl* command uses the OneAPI LevelZero interfaces to identify and mark the existing GPU resources for use by the LWK partition. The following actions using the indicated OneAPI LevelZero interfaces are performed:

1. Set environment variable `ZE_ENABLE_PCI_ID_DEVICE_ORDER` to guarantee consistent device ordering.
2. Find the drivers that are recognized by OneAPI LevelZero using the `zeDriverGet()` interface.
3. Walk through the devices managed by the drivers using the `zeDeviceGet()` interface.
4. Using the `zeDeviceGetProperties()` interface on each found device, look for a device type of `ZE_DEVICE_TYPE_GPU`.
5. Use the `zeDevicePciGetPropertiesExt()` interface to get the domain, bus, device, function of each GPU device.
6. Construct a path to the sysfs device file using the `:domain:bus:device.func` to obtain NUMA id location of the GPU device.
  - Ex. `/sys/bus/pci/devices/0000:93:01.0/numa_node`
7. Use the `zeDeviceGetSubDevices()` interface to get the subdevices (i.e. tiles) of each device.
8. Construct the strings for initializing the mOS sysfs files. For Aurora they will look like the following:
  - `/sys/kernel/mOS/lwkgpus: 0-1,4-5,8-9,12-13,16-17,20-21`
  - `/sys/kernel/mOS/lwkgpus_numa: 0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,1,1,1,1,1,1,1`
  - `/sys/kernel/mOS/lwkgpus_usage_count: 0,0`

Once the GPUs have been designated, they will be available to YOD for resource reservation for the LWK processes. Unlike the core and memory resources, the GPU resources are not explicitly removed from the Linux kernel. This level of isolation was not deemed necessary based on the mechanisms and interfaces that are used to utilize GPUs in an application. Kernel level isolation would have introduced significant modifications to the Linux code base. Unlike cores and memory, the use of GPU resources can be largely controlled by the applications that are allowed to execute in the Linux and LWK partitions.

## GPU Resource Reservation

The reservation of resources for a particular process is done by *yod*. Depending on the *yod* options specified, *yod* will either pass through an existing `ZE_AFFINITY_MASK` environment variable or it will construct one for use by the process. Much of the specific behaviors were described in the previous *PVC mOS User Interface* section. The internal design of *yod* uses the mOS sysfs files that were generated when the LWK partition was created. The *yod* command does not directly use any of the OneAPI LevelZero interfaces. When GPU resources are overcommitted, sharing of GPUs is permitted. The *yod* command will use the `lwkgpus_usage_count` file to balance the sharing across the existing GPU devices. The sharing will be balanced with a NUMA domain for situations when the memory and cores of a process are contained within a NUMA domain.

## Build Configuration

The mOS kernel is built using the standard Linux kernel build procedures. Several scheduler related configuration options should be used when building the mOS kernel. The `*NO_HZ*` and `*RCU_NOCB*` options are not necessary for a low noise mOS scheduler behavior, however building with these configuration options will minimize runtime overhead and minimize noise within the Linux scheduler.

- `CONFIG_MOS_FOR_HPC=y`
- `CONFIG_MOS_LWKM=y`
- `CONFIG_MOS_ONEAPI_LEVEL0=y`  
Needed to enable GPU resource management support.
- `CONFIG_NO_HZ_FULL=y`  
Do not generate regularly scheduled timer wakeups on CPUs designated as `NO_HZ`. On these CPUs, only set timer interrupts for timers explicitly set, for example POSIX timers used within user code.

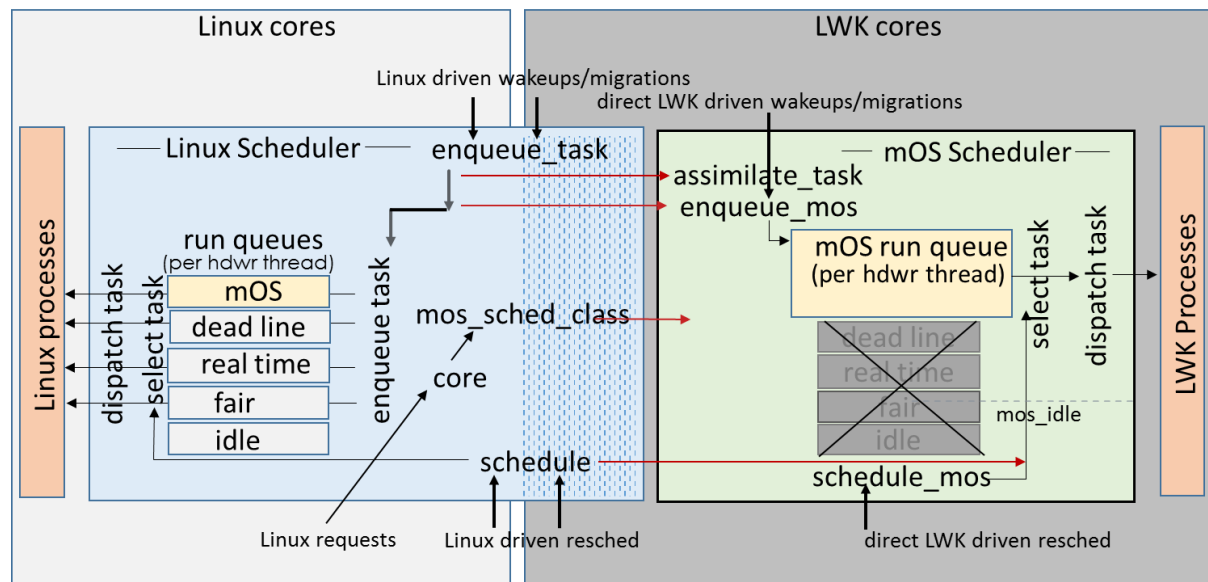
## Boot Parameters

The following boot parameters influence scheduling behaviors and are recommended.

- `intel_pstate=disable`  
This option disables the intel pstate driver and instead uses the standard ACPI P-state driver provided by Linux for management of P-states. This will prevent the processor hardware from actively controlling the P-state. There may be some applications that may benefit by using the intel P-state driver.
- `nmi_watchdog=0`  
When the NMI watchdog is enabled, a non-maskable interrupt will be scheduled on the CPUs at regular intervals. This interrupt handler then checks to see if the CPU appears to be hung. If so, debug information is generated, and specific recovery actions may be taken. In an HPC environment, we do not want the additional interrupt activity generated by this watchdog.
- `nohz_full= 1-<last_cpu_id>`  
This option is used in conjunction with the build configuration `CONFIG_NO_HZ_FULL=y`. It will keep unnecessary interrupts off the CPUs listed. Typically, we will keep all of these kernel housekeeping/timer chores on CPU 0. Optionally the other CPU associated with Core 0 can also be included as a housekeeping CPU if it is determined that Linux requires more resources.
- `kernelcore=16G`  
Restricts the kernel memory to a specific area of physical memory. This will allow for larger contiguous areas of physical memory to be made available to the light-weight kernel partition when a partition is being created.
- `movable_node`  
If on an KNL platform with high bandwidth memory (HBM), this option will allow the light-weight kernel partition to utilize the entire amount of HBM for application memory allocations.

## Kernel block diagram

The following diagram shows the mOS scheduler components and the interaction with the Linux scheduler components. Use this diagram as a reference as you proceed to the subsequent sections.



## Run Queue

A light-weight kernel CPU has one active run queue. That run queue is owned by the mOS scheduler class. This differs from a Linux managed CPU which has a prioritized list of run queues for each scheduler class. The run queues on a Linux CPU are searched in the following order: mOS, dead line, real-time, fair. In the Linux scheduler, tasks with different scheduling policies can be executing on the same CPU, populating multiple run queues. In the mOS scheduler, all tasks run in the mOS scheduling policy. The run queue for the mOS scheduler is composed of two primary objects, a bit mask and a table of list heads. Each bit in this bit mask represents a thread priority level. If a bit position contains a value of '1', then there is at least one runnable thread on the queue at that priority level. When looking for a task to dispatch, the scheduler will determine the bit position of the first bit in the mask having a value of 1. The scheduler then indexes into the table and selects the first thread on the list.

## Assimilation

"Resistance is futile. You will be assimilated". A Linux thread visiting a light-weight kernel CPU will be assimilated. Assimilation is the action of changing that thread's scheduler class from its original class (e.g. "Fair") and placing it into the mOS scheduling class. This will give the mOS scheduler complete control over the scheduling of this task. The task will be dispatched out of the mOS run queue and not out of that task's original run queue. The mOS scheduler will place it at a priority of its choosing. The YOD process itself starts its life as a Linux process. When YOD is ready to transition itself to the light-weight kernel, it sets its own CPU affinity to the light-weight kernel CPUs that it has reserved for the process and then does an 'execvp' to transition itself into the application that it is launching. The migration of the process to the light-weight kernel CPUs is intercepted by a modification to the Linux scheduler, and then the YOD process is assimilated into the mOS scheduler.

There is a short list of additional Linux threads that can run on the light-weight kernel CPUs. These threads are used for very specific purposes that are compatible with the light-weight kernel. These threads are the following:

1. ksoftirqd
2. cpuhp (hot-plug)
3. Any thread in the Linux stop class (e.g. migration helper thread)

The ksoftirqd and cpuhp threads will be assimilated into the mOS scheduler. All other Linux threads are not expected to arrive at an LWK CPU due to the modifications within the Linux scheduler. However, if an unexpected thread does find its way to an LWK CPU, assimilation will occur, and a warning message will be sent to the kernel log.

The threads in the *stop* class are treated specially and will remain in that scheduler class.

### Idle thread

The mOS scheduler has its own idle thread for each light-weight kernel CPU. The idle thread is the thread that executes on the CPU when there is no other work to do. The mOS scheduler will create an idle thread on each LWK CPU when the light-weight kernel partition is created. The thread will exist if that CPU is configured as a light-weight kernel CPU. When an mOS process has the LWK CPU reserved, by default the idle thread will enter MWAIT at C-state=1 when there is no other thread to be dispatched and no interrupts to be processed. The MWAIT will monitor a memory field associated with that CPU. When a thread wishes to wake the CPU, it will perform a store operation to this field, which will end the MWAIT condition. An interrupt will also end the MWAIT condition. If MWAIT is not supported on the platform, the x86 HALT instruction will be used. In this case, an IPI executed from another CPU will be used to wake an idle CPU. The idle mechanism used can be modified by an internal YOD option, '-o idle-control=<mechanism,boundary>'. The idle mechanism can be forced to use either mwait (default), halt, or poll. If poll is selected as the mechanism, the idle thread will spin, polling on a flag in memory that will be set by a thread that wishes to awaken this CPU.

### Core Sleep State

When an mOS process releases its reservation on an LWK CPU, the idle thread will place the LWK CPU into a deeper C-state in order to save power. The mOS scheduler will find the deepest core sleep state supported by the hardware and request that the core is put into this sleep state. Typically, this would be a C-state = 6, resulting in most of the core being powered down. The path to C-state = 6 involves invalidating the TLBs and data caches on the core. All hardware threads in the core must be requesting to enter this level of sleep for the core to reach this C-state. Later, when a new process reserves an LWK CPU, the mOS scheduler will bring the parent core out of deep sleep and place the CPU in and MWAIT at C-state = 1. The boundary condition at which to enter deep sleep can be modified by an internal yod option, '-o idle-control=<mechanism,boundary>'. The valid values for the boundary condition are the following:

- none: Do not impose a boundary for entering deep sleep. Attempt to enter deep sleep immediately when it is determined that there are no threads to be dispatched on this CPU.
- reserved: This is the default behavior. When a CPU is no longer reserved by a process, request to enter deep sleep.
- committed: When a CPU does not have any thread within that process committed to run on it, request to enter deep sleep.

- online: When a CPU is no longer online, attempt to enter deep sleep. By definition, a CPU that is not online is not running the idle loop, therefore we will never attempt to enter deep sleep.

## Round-Robin dispatching

By default, the scheduler does timer-based round-robin dispatching only when a CPU is overcommitted with more than one thread from the application process. If there is only one process thread, the only way that thread is preempted is by the arrival of a higher priority thread. When there is no overcommitment, there are no regularly occurring timer ticks. A block/wakeup sequence or a `sched_yield` system call will cause the next thread on the run queue to be dispatched. It is possible to force the scheduling behavior to FIFO or Round-robin through two different mechanisms:

1. Scheduling policy
2. YOD option

## Scheduling policy

The externally visible scheduling policy for the scheduler defaults to `SCHED_RR`. Instead of introducing a new MOS scheduling policy, the `SCHED_RR` policy is used. This provides compatibility with the pthread library and with various Linux performance and debug tools. The `SCHED_RR` policy is a close match to the default scheduling behavior of the MOS scheduler. A first in first out (FIFO) scheduling policy can be enabled by setting the policy of the thread to `SCHED_FIFO`. This can be done through `pthread_setschedparam`. When the `SCHED_RR` policy is active, if there is more than one equal priority thread on the run queue of a light-weight kernel CPU, the scheduler will run each thread for a maximum time of 100ms before a timer pop will occurring, preempting it and dispatching the next thread on the queue. If the `SCHED_FIFO` policy is active, no timer pop will occur.

## YOD Scheduling policy option

A YOD option can be used to modify the round-robin policy for all threads within the process launched by YOD. This option also gives the caller the ability to override the default time-slice interval to something other than 100ms. This option is specified in the following manner:

```
'-o lwksched-enable-rr=<n>'
```

Each thread will execute up to <n> milliseconds before being preempted by another thread of equal priority. The minimum supported value is 10ms.

## Priority

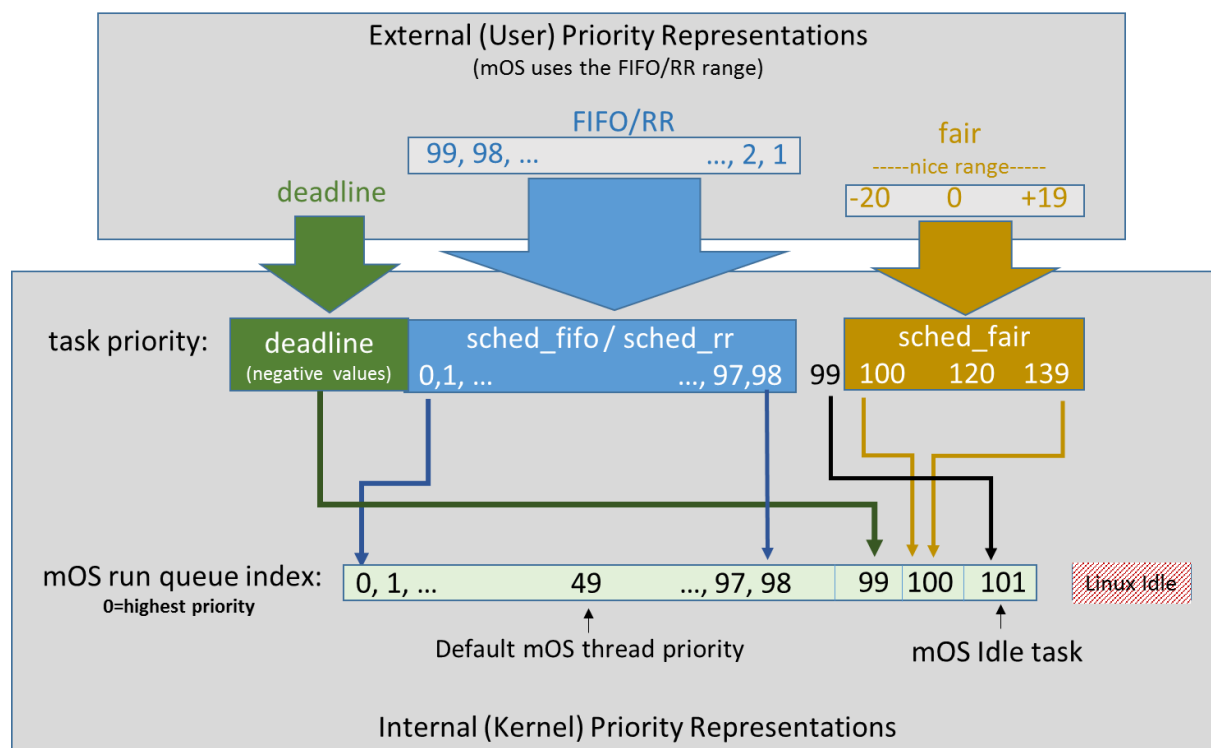
User priorities of 1-99 are supported through standard pthread interfaces such as `pthread_setschedprio`, `pthread_setschedparam`, and `pthread_attr_setschedparam`. When using the `pthread_setschedparam` interface, the `SCHED_FIFO` or `SCHED_RR` policy must be specified along with the priority.

When a job is launched through the YOD job launcher, the kernel will assign a default priority of 50 to the process. When a task that has not been launched through YOD is assimilated into the scheduler (i.e. a Linux task running on the light-weight CPU), it's priority and externally visible policy remains unchanged, however the scheduler will map its priority into an appropriate slot within the scheduler run queue.

The setting of a thread priority is only meaningful when there is more than one thread executing on a CPU. The priority value will determine which thread will be chosen to be dispatched and which task will be preempted. If a high priority task is awakened from a block condition, for example a `futex` wakeup, it

will preempt a lower priority thread running on the thread's home CPU. In many situations, a HPC job will not have more than one thread executing on a CPU, therefore setting priority is not meaningful. The usefulness of setting priority is in situations where over-commitment cannot be avoided or cannot be predicted. Care must be taken when modifying priorities to avoid a hang or deadlock condition. The scheduler will not round-robin threads of different priorities.

### Priority map



### System call `sched_setaffinity` behavior

The `sched_setaffinity` system call when executed from a mOS process will only allow the CPU targets to be LWKCPUs reserved for this process. If the requested target CPU mask contains other CPUs, those CPUs will be excluded from the new CPUs allowed mask. If the resulting mask contains no CPUs, then the system call will return `EINVAL`. If the resulting mask contains at least one valid LWK CPU, the system call will be complete without error. Even though the mOS scheduler may migrate or place utility threads on utility CPUs, user level code is not allowed to explicitly place mOS threads on anything other than an LWK CPU.

When the mOS scheduler initially places or later migrates a utility thread onto a utility CPU, there is no returning to the mOS scheduler. Attempts to use `sched_setaffinity` will return `EINVAL`.

### Light-weight Kernel implemented System Calls

A set of system calls are implemented by the light-weight kernel. This is a list of the system calls implemented by the light-weight kernel:

```
mos_set_clone_attr
mos_mwait
```

### Internal YOD options

The scheduler supports a variety of internal YOD options. These options are not formal options, and may not be available in future versions of MOS. Many of these options are used for experimentation and should be used with caution. These are passed down into the LWK via yod's --opt/-o generalized option passing mechanism. These may also be specified via the YOD\_OPTIONS environment variable (if multiple options are specified, they should be comma delimited).

#### lwksched-disable-setaffinity=<errno>

Do not perform any action as a result of the sched\_setaffinity system call if the target is an mOS thread. On return from the system call, set the returned ERRNO to the value provided.

Examples:

- --opt lwksched-disable-setaffinity=0 will no-op the system call.
- --opt lwksched-disable-setaffinity=38 will cause the system call to fail with ENOSYS

This option is useful for debugging. There are no plans to make it officially supported.

#### lwksched-enable-rr=<time\_quantum>

Enable round-robin dispatching of equal priority threads. Each thread will execute up to <time\_quantum> milliseconds before being preempted by another thread of equal priority. The minimum supported value is 10ms. This option can be used if the application is known to overcommit threads onto CPUs and those threads do not regularly block or yield.

A reasonable value for the time quantum is 100ms. This option should not be used for a typical HPC application that does not over-commit the existing CPUs since it will introduce additional noise.

The current version of the mOS scheduler will automatically set a 100ms round-robin timer if it detects that over-commitment exists on a CPU. The use of this option is not necessary unless the purpose is to modify the time quantum to a value other than the default 100ms.

#### lwksched-stats=<level>

Output counters to the kernel log at the time of process exit. Data detail controlled by the <level>. A value of 1 will generate an entry for every mOS CPU that had more than one mOS thread committed to run on it. A value of 2 will add a summary record for the exiting mOS process. A value of 3 will add records for all CPUs in the process and a process summary record for the exiting process regardless of commitment levels. Information provided:

- PID: The TGID the process. This can be used to visually group the CPUs that belong to a specific process.
- CPUID: CPU corresponding to the data being displayed.
- THREADS: Number of threads within the process (main thread plus pthreads).
- CPUS: Number of CPUs reserved for use by this process.
- MAX\_COMPUTE: High water mark of the number of mOS compute threads assigned to run on this CPU.

- MAX\_UTIL: High water mark of the number of mOS utility threads assigned to run on the CPU.
- CPU MAX\_RUNNING: High water mark of the number of tasks enqueued to the mOS run queue, including kernel tasks.
- GUEST\_DISPATCH: Number of times a non-mOS thread (kernel thread) was dispatched on this CPU.
- TIMER\_POP: The number of timer interrupts. Typically, this would be because of a POSIX timer expiring or RR dispatching if enabled through the option lwksched-enable-rr.
- SETAFFINITY: The number of setaffinity system calls executed by this CPU.
- PUSHED: The number of utility threads that were pushed from this LWK CPU to Linux CPUs to make room for application or runtime worker pthreads.
- BALANCER\_ACTIONS: The number of push/pull balancer actions that have occurred.

#### util-threshold=<X:Y>

The X value indicates the maximum number of LWK CPUs that can be used for hosting utility threads. The Y value represents the maximum number of utility threads allowed to be assigned to any one LWK CPU. Some examples: A value of "0:0" will prevent any utility threads from being placed on an LWK CPU and force all utility threads to be placed on the Linux CPUs that are defined to be the utility CPUs. A value of "-1:1" will allow any number of LWK CPUs to hold utility threads however only a maximum of one utility thread will be assigned to any LWK CPU.

Default behavior is X = -1, Y = 1. This option is useful for experimentation. There are currently no plans to make it officially supported. The UTI API will be the preferred approach to controlling utility thread placement.

#### overcommit-behavior=<behavior>

Set the behavior associated with utility thread CPU placement in situations of CPU over-commitment. This option does not modify the algorithm for selecting a CPU for a compute thread. These behaviors are currently supported:

0 = All Commits: When placing a utility thread, consider the current commitment levels of both compute threads and utility threads when determining the least committed CPU.

1 = Compute Commits: When placing a utility thread, consider only the current commitment levels of compute threads when determining the least committed CPU.

2 = Utility Commits (default): When placing a utility thread, consider only the current commits of utility threads when determining the least committed CPU. This will result in keeping compute threads on their own CPUs and distributing the utility threads evenly across the CPUs used by the compute threads.

There may be some unexpected interactions when also using the util-threshold.

This option is useful for studying behaviors in overcommitted CPU environments that contain utility threads. This over-commitment may be intentional in some application environments. For example, pairing up threads on one CPU that are known not to run at the same time within a phase of the application such as a communication thread and a compute thread.

### one-cpu-per-util

When creating a utility thread to run on a utility CPU, restrict that thread to run on the one CPU with the lowest commit level that satisfies the requested location. If this option is not specified, all the CPUs that satisfy the requested location request will be set in the CPUs allowed mask of the thread, giving the Linux Fair scheduler freedom to balance the workload with the other threads and processes that may be running on these Linux CPUs.

### idle-control=<MECHANISM,BEHAVIOR>

MECHANISM is the fast-path idle/dispatch mechanism used by the idle task. The allowed values are:

- mwait - mwait instruction will be used for both fast dispatch and deep sleep situations
- halt - halt instruction in combination with an IPI sent by the waking thread will be used for fast dispatch. mwait will be used for deep sleep
- poll - polling will be used for fast dispatch, mwait will be used to request deep sleep.

BOUNDARY is the boundary where the fast dispatch mechanism will be deployed. Beyond this boundary, the CPU will request deep sleep. The allowed values are:

- none - Entry to the idle task will always result in a request for deep sleep
- committed - A CPU that has a committed thread to run on it will use the fast dispatch mechanism. A CPU that is reserved by a process but does not have a thread assigned to run on it by the scheduler will request deep sleep using mwait
- reserved - A CPU reserved by the process will use the fast dispatch mechanism. A CPU that is not reserved will request deep sleep using mwait.
- online - All LWK CPUs will use the fast dispatch mechanism. Deep sleep will never be requested.

The default <MECHANISM,BOUNDARY> will be: <mwait,reserved>

### cmci-control=<threshold,poll>

threshold:

- The number of correctable machine check events that must occur in a machine check bank before a correctable machine check interrupt is delivered to a LWK CPU that is hosting an mOS process. The value specified must be less than 32768. A value of 0 disables interrupt delivery to all LWK CPUs hosting the mOS process.
- 'disable-cmci': same as threshold = 0
- 'max-threshold': Set threshold to the maximum allowed value. This will typically be 32767 on newer hardware platforms. (note this is the new default behavior)

poll:

- 'disable-poll': Polling will be disabled on the LWK CPUs that are hosting the mOS process. (note: this was the default behavior before this commit and will remain the default behavior)

'enable-poll': Polling will be enabled and operate with the same interval, behavior, and control as in the Linux OS

In all cases, when the mOS process ends, the machine check banks owned by the associated CPUs will be checked and any pending machine check events will be logged, even if we have not reached the threshold. Also, any disabled CMCI's will be re-enabled, any modified thresholds will be restored, and polling will be re-enabled.

enable-balancer=<balancer\_type><parm1>, <parm2>, <parm3>

'balancer\_type':

- Which balancer should be enabled? Either a 'push' or 'pull' value can be specified. The default value is 'pull'.

'parm1'

- First parameter for the specified balancer type. See Thread Push/Pull Balancers for more details.

'parm2'

- Second parameter for the specified balancer type. See Thread Push/Pull Balancers for more details.

'parm3'

- Third parameter for the specified balancer type. See Thread Push/Pull Balancers for more details.

## Debug Aids

### Counters

During the execution of a process, the scheduler maintains various counters. These counters can be configured to output to the console when the process ends. This is controlled by a YOD option:

'-o lwksched-stats=<n>'

A value of n=1 will generate an entry for every light-weight kernel CPU that has had more than one thread committed to run on it. This provides a straightforward way to identify one of the more common scheduler related performance problems, overcommitting of CPU resources. A value of n=2 will add a summary record for the exiting process. A value of n=3 will add records for all light-weight kernel CPUs in the process and a process summary record for the exiting process regardless of commitment levels. Information provided:

### PID

The TGID the process. This can be used to visually group the CPUs that belong to a specific process.

### CPUID

CPU corresponding to the data being displayed.

### THREADS

Number of threads within the process (main thread plus pthreads).

### CPU

Number of CPUs reserved for use by this process.

### COMPUTE\_COMMIT

High water mark of the number of mOS compute threads assigned to run on this CPU.

### UTIL\_COMMIT

High water mark of the number of mOS utility threads assigned to run on this CPU.

### CPU MAX\_RUNNING

High water mark of the number of threads enqueued to the mOS run queue, including kernel tasks.

### GUEST\_DISPATCH

Number of times a non-mOS thread (kernel thread) was dispatched on this CPU.

### TIMER\_POP

The number of timer interrupts. Typically, this is because of a POSIX timer expiring or RR dispatching if enabled through the option lwksched-enable-rr or the SCHED\_RR policy.

### SETAFFINITY

The number of setaffinity system calls executed by this CPU.

### PUSHED

The number of utility threads that were pushed from this LWK CPU to Linux CPUs to make room for application or runtime worker pthreads.

### BALANCER\_ACTIONS

The number of push or pull balancer actions that have occurred.

## Trace-point Events

The scheduler has instrumented many trace records within the Linux kernel's ftrace/trace-point framework (/sys/kernel/debug/tracing). This framework allows very flexible filtering and enabling options. Enabling specific mOS tracing along with a very large selection of Linux trace-points can be helpful in the analysis of problems. Every trace entry contains a set of common data with the following format:

```
#          _-----> irqs-off
#          /_-----> need-resched
#          | /_-----> hardirq/softirq
#          || /_----> preempt-depth
#          ||| /_    delay
#          TASK-PID  CPU#  | | |  TIMESTAMP  FUNCTION
#          | |      |   | | |  |          |
# migration/67-291  [067] d..1 98743.136164: my_trace_point
```

The mOS trace points grouped in two directories:

- `/sys/kernel/debug/tracing/events/mos/`
- `/sys/kernel/debug/tracing/events/mos_idle/`

Most of the events are in the `.../mos/` directory. The events in the `.../mos_idle/` directory may be generated at an extremely high rate and should therefore have filtering applied. These events are in their own directory so that all other events can be conveniently enabled without filtering. For example, enabling all of the events in `.../mos/` can be accomplished with:

```
echo 1 > /sys/kernel/debug/tracing/events/mos/enable
```

For a description on how to filter and enable events, refer to the following text file in the documentation directory of the Linux kernel source: `.../Documentation/trace/ftrace.txt`.

This is the list of mOS trace points that can be enabled in `/sys/kernel/debug/tracing/events/mos/`:

#### [\*mos\\_clone\\_cpu\\_assign\*](#)

The scheduler selected a CPU for a new thread being created by the clone system call.

This trace-point provides the following additional data:

- `target cpu`: The CPU on which the new thread will be started.
- `pid`: the PID of the new thread.
- `num cpus allowed`: the number of CPUs within in the `cpus_allowed` mask for the current task.

Example:

```
mos_clone_cpu_assign: target cpu=20 pid=44602 num cpus allowed=32
```

#### [\*mos\\_timer\\_tick\*](#)

A scheduler timer tick interrupt was handled.

This trace-point provides the following additional data:

- `policy`: The scheduling policy in effect for the current task.
- `num cpus allowed`: the number of CPUs within the `cpus_allowed` mask for the current task.

Example:

```
mos_timer_tick: policy=0 num cpus allowed=1
```

#### [\*mos\\_idle\\_init\*](#)

An idle thread is being created for an LWKCPU.

This trace-point provides the following additional data:

- `cpu`: The CPU for which this idle thread is being created.

Example:

```
mos_idle_init: create for cpu=20
```

#### *mos\_cpu\_commit*

A commit is being recorded for an mOS thread on a CPU.

This trace-point provides the following additional data:

- pid: The PID of the thread holding the new commit.
- cpu: The CPU that the thread is being committed to.
- compute commits: resulting count of compute commits on this CPU.
- utility commits: resulting count of utility commits on this CPU.

Example:

```
mos_cpu_commit: pid=44618 cpu=24 compute_level=1 util_level=0
```

#### *mos\_cpu\_uncommit*

A commit is being released.

This trace-point provides the following additional data:

- pid: The PID of the thread releasing the commit.
- cpu: The CPU that the commit is being released from.
- compute commits: resulting count of compute commits on this CPU.
- utility commits: resulting count of utility commits on the CPU.

Example:

```
mos_cpu_uncommit: pid=44570 cpu=63 compute_level=0 util_level=0
```

#### *mos\_cpu\_select\_unavail*

A CPU matching the requested filters and limits could not be found.

This trace-point provides the following additional data:

- pid: The PID of the thread for whom we are searching.
- commit type: What type of commits used for commit level testing.
  - 0: All commits
  - 1: Compute commits
  - 2: Utility commits
- commit lvl: The maximum allowed commit level for this search.
- match type: What type of search.
  - 0: First Available
  - 1: Same Core
  - 2: Same L1
  - 3: Same L2
  - 4: Same L3
  - 5: Same Domain

- 6: Other Core
- 7: Other L1
- 8: Other L2
- 9: Other L3
- 10: Other Domain
- 11: In Node Mask
- match id: The value depends on the type of match. . Not applicable for match type 0.
- range: How many of the LWKCPUs in the process should be searched. A value of -1 means to search all the available CPUs.
- excl pid: If the request is to run exclusively on a CPU, this is the requesting PID.

Example:

```
mos_cpu_select_unavail: pid=44949 requested commit level=0 type=0 id=0
range=-1
```

*mos\_cpu\_select*

A CPU matching the requested filters and limits was found.

This trace-point provides the following additional data:

- pid: The PID of the thread for whom we are searching.
- cpu: The selected CPU.
- commit type: What type of commits used for commit level testing.
  - 0: All commits
  - 1: Compute commits
  - 2: Utility commits
- commit lvl: The maximum allowed commit level for this search.
- match type: What type of search.
  - 0: First Available
  - 1: Same Core
  - 2: Same L1
  - 3: Same L2
  - 4: Same L3
  - 5: Same Domain
  - 6: Other Core
  - 7: Other L1
  - 8: Other L2
  - 9: Other L3
  - 10: Other Domain
  - 11: In Node Mask
- match id: The value depends on the type of match. Not applicable for match type 0.
- range: How many of the LWKCPUs in the process should be searched. A value of -1 means to search all the available CPUs.
- excl pid: If the request is to run exclusively on a CPU, this is the requesting PID.

Example:

```
mos_cpu_select: pid=17784 cpu=45 commit type=0 commit lvl=1 match type=0  
match id=-1 range=-1 excl pid=0
```

*mos\_util\_thread\_assigned*

A utility thread was assigned to a CPU.

This trace-point provides the following additional data:

- **cpu:** The CPU assigned to the utility thread.
- **placement:** was requested location information honored by the scheduler.

Example:

```
mos_util_thread_assigned: cpu=45 placement_honored=1
```

*mos\_util\_thread\_pushed*

A utility thread was pushed from an LWKCPU to a Linux CPU to make room for a compute thread.

This trace-point provides the following additional data:

- **pid:** This PID of the thread that is being pushed.
- **from:** The CPU that will no longer be assigned to this thread.
- **to:** The CPU currently hosting this thread.
- **commit:** The current utility thread commit level of the CPU now hosting this thread.
- **placement honored:** If no placement was originally requested or if placement was requested and we were able to also satisfy the request on the Linux CPU, then this will be set to '1'. Otherwise '0'.

Example:

```
mos_util_thread_pushed: pid=47897 from=61 to=19 commit=0 placement_honored=1
```

*mos\_assimilate\_launch*

The YOD job launcher has placed itself on an LWKCPU in preparation for the execvp to being executing the target job. This has resulted in mOS assimilating this thread into the mOS scheduler.

This trace-point provides the following additional data:

- **comm:** Task name. In the normal situation of YOD has the job launcher, this will be "yod"
- **cpu:** The CPU on which the thread as been assigned to run on.
- **policy:** The scheduling policy that the thread will be running with. SCHED\_NORMAL=0, SCHED\_FIFO=1, SCHED\_RR=2, SCHED\_DEADLINE=6.
- **type:** the mOS thread type: normal=0, utility=1, idle=2, guest=3.
- **nr\_cpus:** how many CPUs are in the cpus\_allowed mask for this task.

Example:

```
mos_assimilate_launch: comm=yod cpu=2 policy=1 type=0 nr_cpus=32
```

#### *mos\_clone\_attr\_active*

An mOS clone attribute system call was executed that has generated pending clone attributes to be used during the next clone system call within this thread.

This trace-point provides the following additional data:

- behavior: utility thread behavior request as defined in `.../include/uapi/linux/mos.h`.
- placement: utility thread location request as defined in `.../include/uapi/linux/mos.h`.

Example:

```
mos_clone_attr_active: behavior=1 placement=512
```

#### *mos\_clone\_attr\_cleared*

An mOS clone attribute system call was executed that has cleared any pending clone attributes. The next clone system call will not use these attributes.

This trace-point provides the following additional data:

- behavior: previous utility thread behavior request as defined in `.../include/uapi/linux/mos.h`.
- placement: previous utility thread location request as defined in `.../include/uapi/linux/mos.h`.

Example:

```
mos_clone_attr_cleared: previous behavior=16 placement=64
```

#### *mos\_assimilate\_guest*

A thread that is not associated with the target application has entered the mOS schedule and has been assimilated. It will now abide by the mOS scheduler rules.

This trace-point provides the following additional data:

- comm: Name of the task being assimilated.
- cpu: The CPU on which it is being assimilated.
- policy: The scheduling policy of the task being assimilated. SCHED\_NORMAL=0, SCHED\_FIFO=1, SCHED\_RR=2, SCHED\_DEADLINE=6.
- type: the mOS thread type: normal=0, utility=1, idle=2, guest=3.
- nr\_cpus: how many CPUs are in the cpus\_allowed mask for this task.

Example:

```
mos_assimilate_guest: comm=cpuhp/2 cpu=2 policy=0 type=3 nr_cpus=1
```

#### *mos\_assimilate\_idle*

An mOS idle thread has been created and is now being assimilated. This action occurs once for each CPU the first time an application thread begins to run on an LWK CPU after the partition has been created. The idle task will remain in place on this CPU until the LWK partition is destroyed.

This trace-point provides the following additional data:

- comm: Name of the task being assimilated.
- cpu: The CPU on which it is being assimilated.
- policy: The scheduling policy of the task being assimilated. SCHED\_NORMAL=0, SCHED\_FIFO=1, SCHED\_RR=2, SCHED\_DEADLINE=6.
- type: the mOS thread type: normal=0, utility=1, idle=2, guest=3.
- nr\_cpus: how many CPUs are in the cpus\_allowed mask for this task.

Example:

```
mos_assimilate_idle: comm=mos_idle/27 cpu=27 policy=0 type=2 nr_cpus=1
```

#### *mos\_giveback\_thread*

An LWK partition has been deleted and we have given back this assimilated thread to the Linux scheduler.

This trace-point provides the following additional data:

- comm: Name of the task being returned to Linux.
- cpu: The CPU on which this giveback is occurring.
- policy: The scheduling policy of the task being returned. SCHED\_NORMAL=0, SCHED\_FIFO=1, SCHED\_RR=2, SCHED\_DEADLINE=6.
- type: the mOS thread type: normal=0, utility=1, idle=2, guest=3.
- nr\_cpus: how many CPUs are in the cpus\_allowed mask for this task.

Example:

```
mos_giveback_thread: comm=cpuhp/41 cpu=41 policy=0 type=3 nr_cpus=1
```

#### *mos\_select\_main\_thread\_home*

The main thread was not executing on its preferred original CPU and the cpus\_allowed mask for the main thread was changed to again allow execution on this CPU. This CPU will be selected as the home for the main thread once again.

This trace-point provides the following additional data:

- pid: The PID of the main thread for which a new CPU has been selected.
- from\_cpu: The current home CPU of the main thread.
- to\_cpu: The selected new (original) home CPU for the main thread.

Example:

```
mos_select_main_thread_home: pid=15629 from cpu=62 to cpu=2
```

#### [mos\\_mwait\\_cstates\\_configured](#)

The hardware has been probed and the minimum and maximum C-state values have been determined. This C-state information will be used by the idle tasks.

This trace-point provides the following additional data:

- min C-state
- max C-state
- ecx
- substates

Example:

```
mos_mwait_cstates_configured: hints min=80000000 max=c0000010 ecx=00000003  
substates=00000110
```

This is the list of mOS trace points that can be enabled in `/sys/kernel/debug/tracing/events/mos_idle/`:

#### [mos\\_mwait\\_idle\\_entry](#)

The idle thread is entering MWAIT.

This trace-point provides the following additional data:

- ecx register
- eax register

Example:

```
mos_mwait_idle_entry: Enter MWAIT... ecx=00000001 eax=00000010
```

#### [mos\\_mwait\\_idle\\_exit](#)

The idle thread is exiting MWAIT.

This trace-point provides the following additional data:

- ecx register
- eax register

Example:

```
mos_mwait_idle_exit: ...Leave MWAIT ecx=00000001 eax=00000010
```

#### [mos\\_halt\\_idle\\_entry](#)

The idle thread is entering the halt instruction.

This trace-point provides the following additional data:

- (none)

Example:

```
mos_halt_idle_entry: Enter HALT...
```

#### *mos\_halt\_idle\_exit*

The idle thread is exiting HALT.

This trace-point provides the following additional data:

- (none)

Example:

```
mos_halt_idle_exit: ...Leave HALT
```

#### *mos\_poll\_idle\_entry*

The idle thread is beginning to poll.

This trace-point provides the following additional data:

- (none)

Example:

```
mos_poll_idle_entry: Enter POLL...
```

#### *mos\_poll\_idle\_exit*

The idle thread is ending its polling.

This trace-point provides the following additional data:

- (none)

Example:

```
mos_poll_idle_exit: ...Leave POLL
```

#### *mos\_balancer\_tick\_start*

The timer for PULL balancer idle tick is being started.

This trace-point provides the following additional data:

- Period of the timer tick in milliseconds

Example:

```
mos_balancer_tick_start: period(ms)=100
```

#### *mos\_balancer\_tick\_stop*

The timer for the PULL balancer idle tick is being stopped. This will occur when exiting idle.

This trace-point provides the following additional data:

- Period of the timer tick in milliseconds

Example:

```
mos_balancer_tick_stop: period(ms)=100
```

### *mos\_balancer\_tick*

The PULL balancer timer tick has occurred, and we are running in the handler.

This trace-point provides the following additional data:

- Period of the timer tick in milliseconds

Example:

```
mos_balancer_tick: current period(ms)=100
```

### *mos\_balance*

The balancer has moved a thread from one CPU to another CPU.

This trace-point provides the following additional data:

- PID of the thread being moved
- CPU ID of the previous run queue
- CPU ID of the new run queue
- Load of the previous CPU
- Load of the new CPU

## Set Affinity Disable

A class of problems that can occur when executing an HPC application in mOS involves conflicts caused when the application or runtime attempts to explicitly set threads on specific CPUs. The mOS kernel tries to control the placement of its processes and associated threads but will respect any valid request to place a thread on specific CPUs using the `sched_setaffinity` system call. As previously mentioned, a count of how many `sched_setaffinity` system calls have been executed per thread can be surfaced when the job ends using the `lwksched-stats` option. This will give an indication on how much `sched_setaffinity` actions are being done by the application or runtime. Also using the `ftrace` facility, more detail information regarding the execution of specific `sched_setaffinity` system calls can be extracted. However, if a more active approach to analyzing a problem related to setting of affinity is desired, the following YOD option can be used:

```
'-o lwksched-disable-setaffinity=<errno>'
```

This option will disable the execution of a `sched-setaffinity` system call. The system call will return the `errno` value that is specified in the YOD option.

Examples:

- `'-o lwksched-disable-setaffinity=0'` will no-op the system call, appearing to the caller as successful.
- `'-o lwksched-disable-setaffinity=38'` will cause the system call to fail with `ENOSYS`.

## Appendix

### Utility Thread Application Programmer's Interface

The Utility thread API (UTI API) has been jointly developed by RIKEN Advanced Institute for Computational Science - Japan, Sandia National Laboratories, and Intel Corp. with feedback from Fujitsu and Argonne National Laboratory. The mOS scheduler fully supports this API. The information in this section explains the usage of this interface and the capabilities that it provides.

#### Why use this API?

The UTI API allows run-times and applications to control the placement of the threads which are not the primary computational threads within the application.

The API:

- Keeps these extra threads from interfering with computational threads.
- Allows grouping and placing of utility threads across the ranks within a node to maximize performance.
- Does not require the caller to have detailed knowledge of the system topology or the scheduler. Allows the kernel to provide intelligent placement and scheduling behavior.
- Does not require the caller to be aware of other potentially conflicting run-time or application thread placement actions. CPU selection is managed globally across the node by mOS.

#### Utility Thread API Compiling and Linking

Header file `/usr/include/uti.h` contains the function and macro declarations.

- `#include <uti.h>`

Library `/usr/lib/libmos.so` contains the mOS implementation of the UTI API. Link using the following:

- `"-lmos"`

#### Linux compatibility

In the future, when a Linux implementation of the UTI is available, there will be no need to explicitly link with `"-lmos"`. Linking with the Linux library at compile time would be sufficient. When launching an mOS job through YOD, the `libmos.so` will be preloaded by YOD, transparently replacing the Linux implementation with the mOS implementation.

#### Behaviors and Locations

The programmer can provide behavior and location hints to the kernel. The kernel will then use its knowledge of the system topology and available scheduling facilities to intelligently place and run the utility thread. The following lists show the different behavior and location attributes that can be provided.

#### Behavior

Specifying a behavior allows the scheduler to optimize scheduling actions for the utility thread.

- CPU intensive, e.g., constant polling.
- Expects high scheduling priority.

- Expects low scheduling priority.
- Does not play nice with others; infrequent yields and/or blocks.
- Expects to run on a dedicated CPU.

#### Locations

- Place on various combinations of same/different L1/L2/L3/NUMA-domain.
- Place on a specific NUMA domain.
- Place on a Linux CPU.
- Place on a lightweight kernel CPU.
- Place on CPUs handling fabric interrupts.

#### Specifying a Location

There are several ways of specifying a location:

- Explicit NUMA domain
  - Supply a bit mask containing NUMA domains.
- Location relative to the caller of the API.
  - Same L1, L2, L3, or NUMA domain
  - Different L1, L2, L3, or NUMA domain
- Location relative to other utility threads specifying a common key.
  - Allows grouping of utility threads used across ranks within the node.
  - Used in conjunction with a specification of "Same L1, L2, L3, or NUMA domain"
- Type of CPU
  - Can be used in conjunction with the above location specifications
  - FWK - Linux CPU running under the Linux scheduler
  - LWK - lightweight kernel-controlled CPU
  - Fabric Interrupt handling CPU

#### Location Key Example

This example shows the required sequence of operations to place utility threads on Linux CPUs running under the same L2 cache.

- Run-time agrees on a unique key value to use across ranks within a node.
- Each rank creates a utility thread and specifies:
  - The same location key value.
  - Request *Same L2*
  - Request *FWK* CPU type
- When the first utility thread is created, mOS will pick an appropriate Linux CPU and L2 cache.
- All subsequent utility threads created with the same key will be placed on Linux CPUs and share the same L2 cache.
- The mOS kernel will assign the utility threads balanced across the available CPUs that satisfy the location requested.

#### UTI Attribute Object

The UTI attribute object is an opaque object that contains the behavior and location information to be used by the kernel when a pthread is created. The definition of the fields within the object are OS specific and purposely hidden from the user interface. This object is treated similarly to the

pthread\_attr object within the pthread library. This object is passed to the uti\_pthread\_create() interface, along with the standard arguments passed to pthread\_create(). The libmos.so library contains the functions used to prepare the attribute object for use.

#### *Initializing and Destroying the Attribute Object*

The following function is provided for initializing the attribute object before use:

- `int uti_attr_init(uti_attr_t *attr);`

The following function is provided to destroy the attribute object:

- `int uti_attr_destroy(uti_attr_t *attr)`

#### *Setting behaviors*

This is the list of library functions used to set behaviors in the attribute object:

- `int uti_attr_cpu_intensive(uti_attr_t *attr);`
  - CPU intensive thread, e.g. constant polling
- `int uti_attr_high_priority(uti_attr_t *attr);`
  - Expects high scheduling priority
- `int uti_attr_low_priority(uti_attr_t *attr)`
  - Expects low scheduling priority
- `int uti_attr_non_cooperative(uti_attr_t *attr)`
  - Does not play nice with others. Infrequent yields and/or blocks
- `int uti_attr_exclusive_cpu(uti_attr_t *attr);`
  - Expects to run on a dedicated CPU

#### *Setting locations*

This is the list of library functions used to set location in the attribute object:

- `int uti_attr_numa_set(uti_attr_t *attr, unsigned long *nodemask, unsigned long maxnodes);`
- `int uti_attr_same_numa_domain(uti_attr_t *attr);`
- `int uti_attr_different_numa_domain(uti_attr_t *attr);`
- `int uti_attr_same_l1(uti_attr_t *attr);`
- `int uti_attr_different_l1(uti_attr_t *attr);`
- `int uti_attr_same_l2(uti_attr_t *attr);`
- `int uti_attr_different_l2(uti_attr_t *attr);`
- `int uti_attr_same_l3(uti_attr_t *attr);`
- `int uti_attr_different_l3(uti_attr_t *attr);`
- `int uti_attr_prefer_lwk(uti_attr_t *attr);`
- `int uti_attr_prefer_fwkw(uti_attr_t *attr);`
- `int uti_attr_fabric_intr_affinity(uti_attr_t *attr);`
- `int uti_attr_location_key(uti_attr_t *attr, unsigned long key);`

#### *Determining results*

The uti\_pthread\_create interface will return EINVAL if conflicting or invalid specifications are provided in the UTI attributes. For example, EINVAL will be returned if 'Same L2' and 'Different L2' are both requested. In these cases, no thread will be created. In other situations when there is no obvious

conflict, the thread will be created, even if the requested location or behavior cannot be satisfied. Location and behavior results can be determined using the interfaces listed below. The return values are 1=true, and 0=false. The setting of pthread attributes should be used with caution since they will override the actions/results provided by the UTI attributes.

- int uti\_result\_different\_numa\_domain(uti\_attr\_t \*attr);
- int uti\_result\_same\_l1(uti\_attr\_t \*attr);
- int uti\_result\_different\_l1(uti\_attr\_t \*attr);
- int uti\_result\_same\_l2(uti\_attr\_t \*attr);
- int uti\_result\_different\_l2(uti\_attr\_t \*attr);
- int uti\_result\_same\_l3(uti\_attr\_t \*attr);
- int uti\_result\_different\_l3(uti\_attr\_t \*attr);
- int uti\_result\_prefer\_lwk(uti\_attr\_t \*attr);
- int uti\_result\_prefer\_fwk(uti\_attr\_t \*attr);
- int uti\_result\_fabric\_intr\_affinity(uti\_attr\_t \*attr);
- int uti\_result\_exclusive\_cpu(uti\_attr\_t \*attr);
- int uti\_result\_cpu\_intensive(uti\_attr\_t \*attr);
- int uti\_result\_high\_priority(uti\_attr\_t \*attr);
- int uti\_result\_low\_priority(uti\_attr\_t \*attr);
- int uti\_result\_non\_cooperative(uti\_attr\_t \*attr);
- int uti\_result\_location(uti\_attr\_t \*attr);
- int uti\_result\_behavior(uti\_attr\_t \*attr);
- int uti\_result(uti\_attr\_t \*attr);

### Usage Example

Note: if your application could be running concurrently with another application using the UTI API, you may need to generate a location key that does not mistakenly match the key in the other application. This example simply uses a statically defined key value.

```
#include <uti.h>
pthread_attr_t p_attr;
uti_attr_t uti_attr;
int ret;
..
/* Initialize the attribute objects */
if ((ret = pthread_attr_init(&p_attr)) ||
    (ret = uti_attr_init(&uti_attr)))
    goto uti_exit;
/* Request to put the thread on the same L2 as other utility threads.
 * Also indicate that the thread repeatedly monitors a device.
 */
if ((ret = uti_attr_same_l2(&uti_attr)) ||
    (ret = uti_attr_location_key(&uti_attr, 123456)) ||
    (ret = uti_attr_cpu_intensive(&uti_attr)))
    goto uti_exit;
```

```

/* Create the utility thread */
if ((ret = uti_pthread_create(idp, &p_attr, thread_start, thread_info, &uti_attr)))
    goto uti_exit;
/* Did the system accept our location and behavior request */
if (!uti_result(&uti_attr))
    printf("Warning: utility thread attributes not honored.\n");
if ((ret = uti_attr_destroy(&uti_attr)))
    goto uti_exit;
..
uti_exit;;

```

#### *Interactions between pthread\_attr and uti\_attr*

Avoid the use of pthread\_attr\_setaffinity\_np when specifying a location with the uti\_attr object. The pthread\_attr\_setaffinity\_np directive is prioritized over the uti\_attr location requests. If valid CPUs are specified, this action may alter the placement directives requested by the UTI attributes object. If invalid CPUs are provided, this will result in the uti\_pthread\_create interface returning EINVAL with no utility thread created.

Avoid the use of pthread\_attr\_set\_schedparam and pthread\_attr\_setschedpolicy when specifying a behavior within the uti\_attr object. These attributes are prioritized over the uti\_attr behavior requests. Usage may alter the actions that would have been taken based on the uti\_attributes behavior hints. A policy or param that is invalid for an mOS process will result in the uti\_pthread\_create interface returning EINVAL with no utility thread created.