CS 60038: Advances in Operating Systems Design

MACH Overview

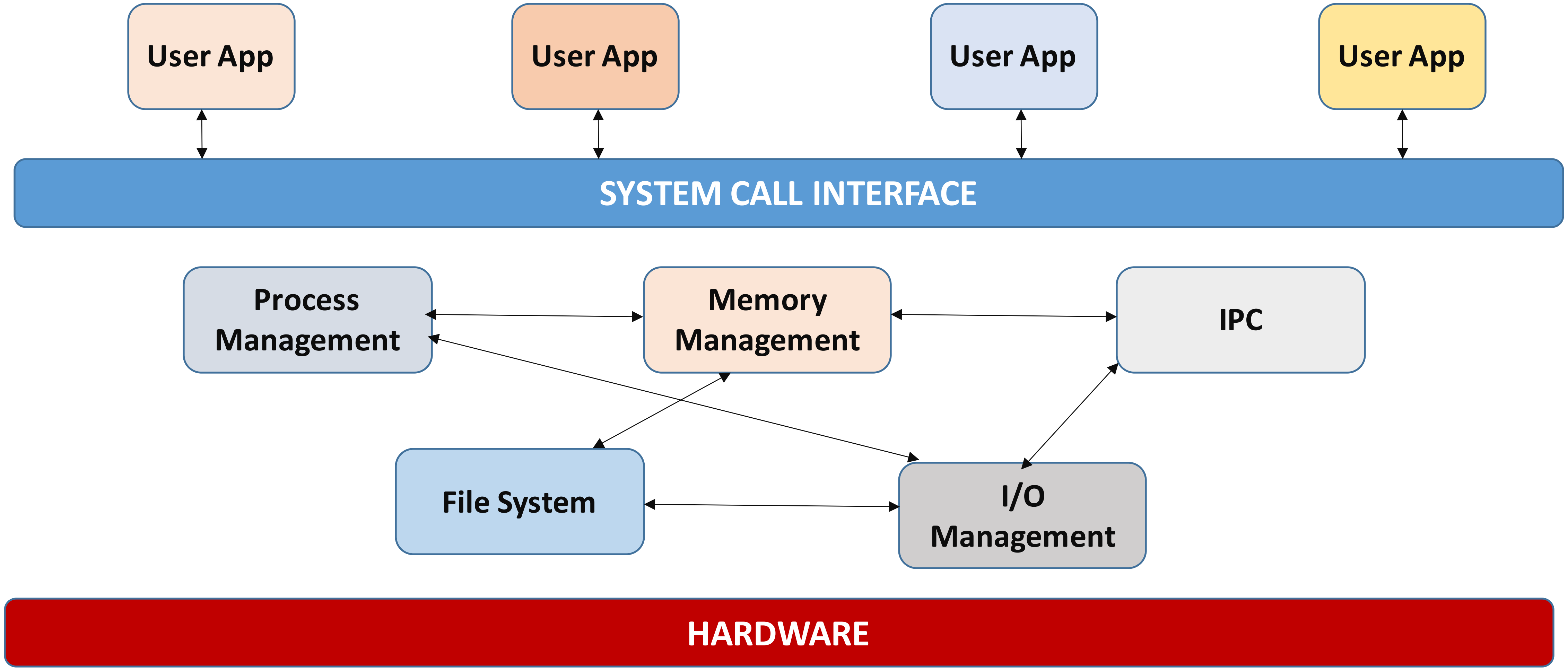
Kenel Structure and Task Scheduling

Department of Computer Science and Engineering

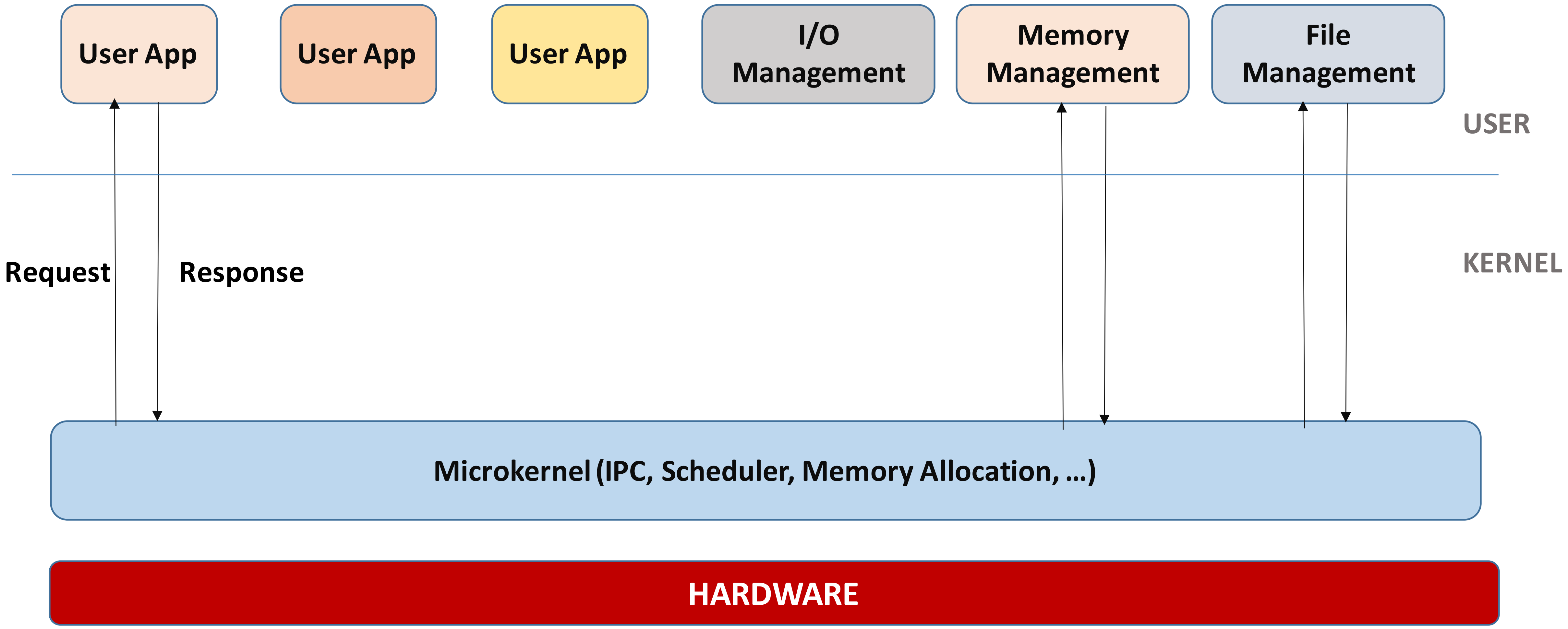
INDIAN INSTITUTE OF TECHNOLOGY KHARAGPUR

Sandip Chakraborty
sandipc@cse.iitkgp.ac.in

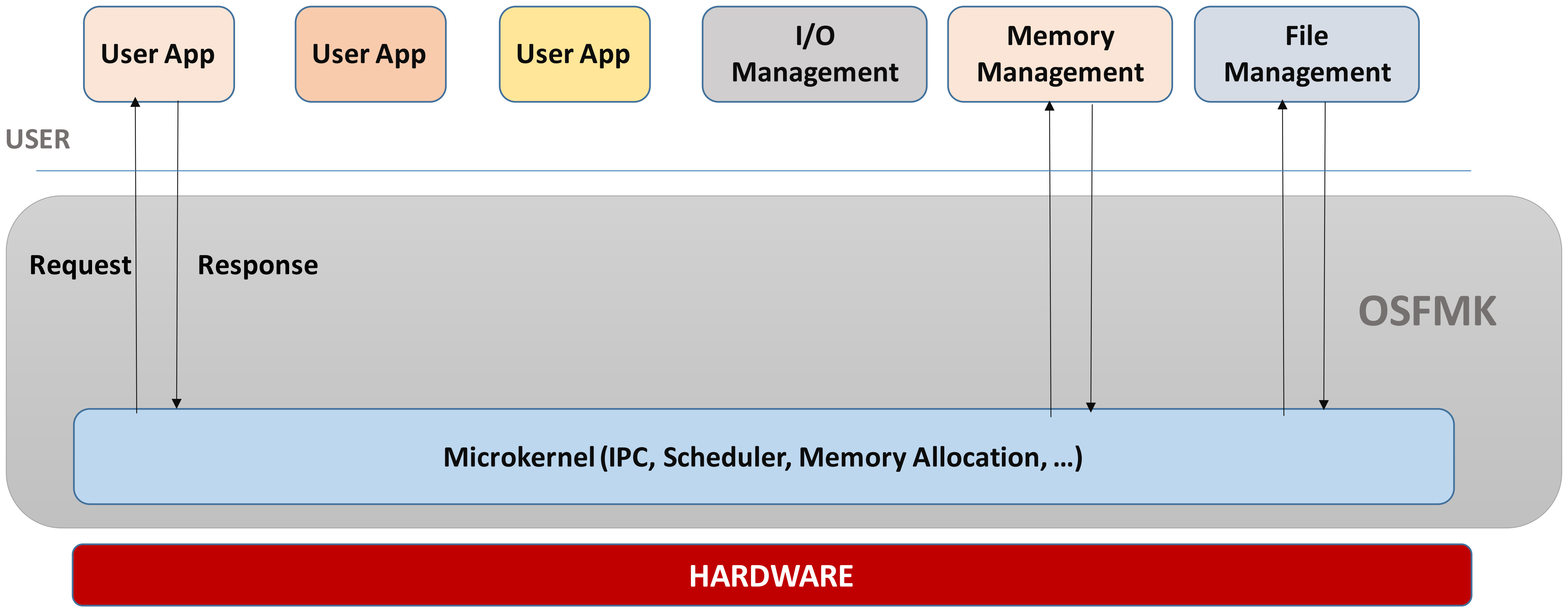
Monolithic Kernel



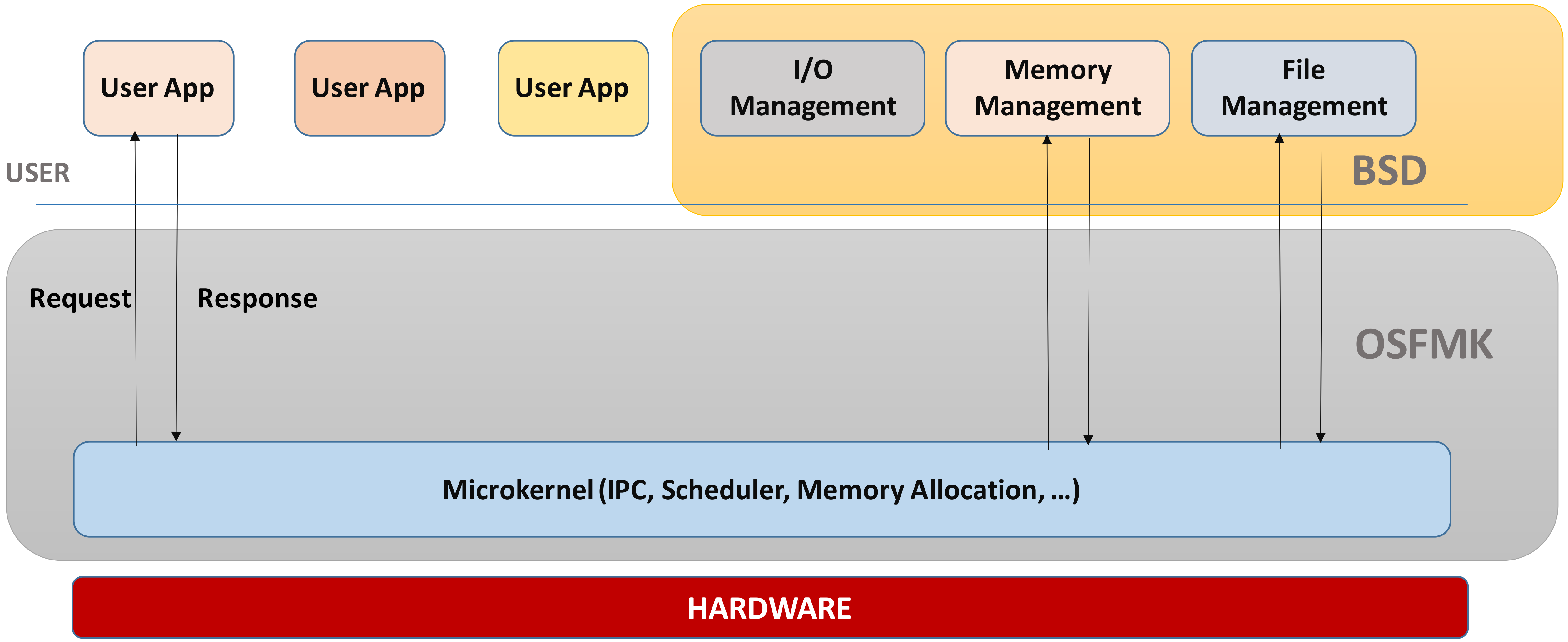
Microkernel



XNU Abstraction

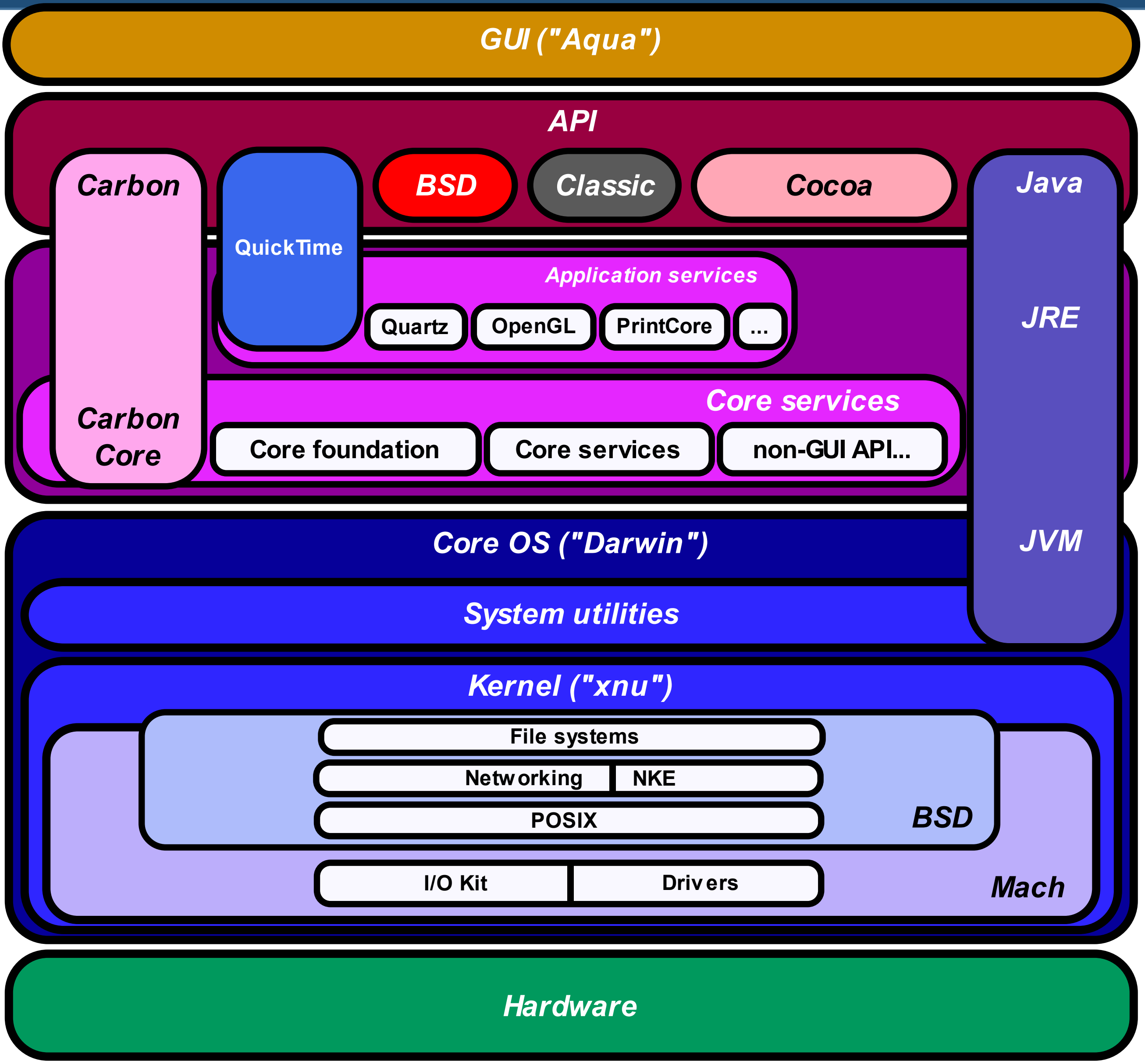


XNU Abstraction



The Darwin OS -- the Core of Apple OS

audioOS - Homepod
macOS - iMac
bridgeOS - Apple HW
iOS - iPhone/iPad
tvOS - Apple TV
watchOS - Apple Watch



MACH Primitives

- It has a highly minimalist concept:
  - A thin, minimal core, supporting an object-oriented model wherein individual, well-defined components (in effect, subsystems) communicate with one another by means of messages.
- Unlike other OS, which present a complete model on top of which user mode processes may be implemented, Mach provides a bare-bones model, on top of which the OS itself may be implemented.
- Mach is, essentially, a kernel-within-a kernel.
  - While Mach calls are visible from user mode, they implement a deep core, on top of which a larger kernel may be implemented.
- OS X's XNU is one specific implementation of UNIX (specifically, BSD) over Mach

XNU Source: https://opensource.apple.com/source/xnu/

Everything is Object

- In Mach, everything is implemented as its own object.
- Processes (which Mach calls tasks), threads, and virtual memory are objects, each with its own properties.
- Mach implements object-to-object communication by means of message passing.
- Unlike other architecture, Mach objects cannot directly invoke or call on one another, rather they are required to pass messages.

Message Passing

- The source object sends a message, which is queued by the target object until it can be processed and handled.
- Similarly, the message processing may produce a reply, which is sent back by means of a separate message
- Messages are delivered reliably in a FIFO manner.
- The content of the message is entirely up to the sender and the receiver to negotiate.

Mach Kernel Abstractions

- All functionalities are moved out of the kernel into user mode, leaving the kernel with the bare minima.
- Mach provides a small set of abstractions that have been designed to be both simple and powerful.

Mach Kernel Abstractions

- All functionalities are moved out of the kernel into user mode, leaving the kernel with the bare minima.
- Mach provides a small set of abstractions that have been designed to be both simple and powerful.

Mach Kernel Abstractions

- These are the main kernel abstractions:

Mach Kernel Abstractions

- These are the main kernel abstractions:
  - **Tasks -** The units of resource ownership; each task consists of a virtual address space, a port right namespace, and one or more threads(Similar to a process.). Allocation of resources to individual threads or groups

Mach Kernel Abstractions

- These are the main kernel abstractions:
  - **Tasks -** The units of resource ownership; each task consists of a virtual address space, a port right namespace, and one or more threads(Similar to a process.). Allocation of resources to individual threads or groups
  - **Threads** - The units of CPU execution within a task.

Mach Kernel Abstractions

- These are the main kernel abstractions:
  - **Tasks -** The units of resource ownership; each task consists of a virtual address space, a port right namespace, and one or more threads(Similar to a process.). Allocation of resources to individual threads or groups
  - **Threads** - The units of CPU execution within a task.
  - **Address space -** In conjunction with memory managers, Mach implements the notion of a sparse virtual address space and shared memory.

Mach Kernel Abstractions

- These are the main kernel abstractions:
  - **Tasks -** The units of resource ownership; each task consists of a virtual address space, a port right namespace, and one or more threads(Similar to a process.). Allocation of resources to individual threads or groups
  - **Threads** - The units of CPU execution within a task.
  - **Address space -** In conjunction with memory managers, Mach implements the notion of a sparse virtual address space and shared memory.
  - **Memory objects** - The internal units of memory management. Memory objects include named entries and regions; they are representations of potentially persistent data that may be mapped into address spaces.

Mach Kernel Abstractions

- These are the main kernel abstractions:
  - **Tasks -** The units of resource ownership; each task consists of a virtual address space, a port right namespace, and one or more threads(Similar to a process.). Allocation of resources to individual threads or groups
  - **Threads** - The units of CPU execution within a task.
  - **Address space -** In conjunction with memory managers, Mach implements the notion of a sparse virtual address space and shared memory.
  - **Memory objects** - The internal units of memory management. Memory objects include named entries and regions; they are representations of potentially persistent data that may be mapped into address spaces.
  - **Ports -** Secure, simplex communication channels, accessible only via send and receive capabilities (known as port rights).

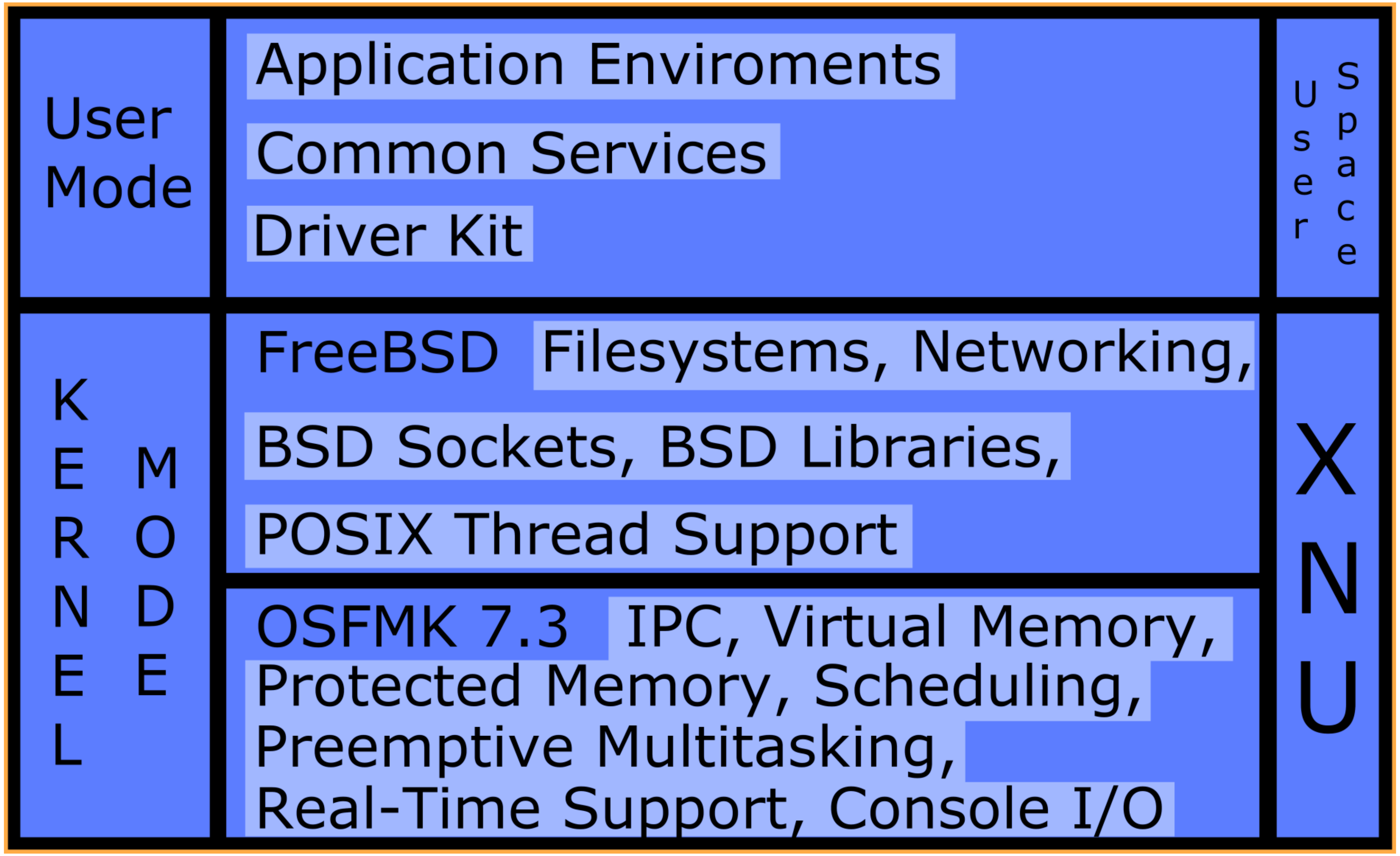
Mach Kernel Abstractions

- These are the main kernel abstractions:
  - **Tasks -** The units of resource ownership; each task consists of a virtual address space, a port right namespace, and one or more threads(Similar to a process.). Allocation of resources to individual threads or groups
  - **Threads** - The units of CPU execution within a task.
  - **Address space -** In conjunction with memory managers, Mach implements the notion of a sparse virtual address space and shared memory.
  - **Memory objects** - The internal units of memory management. Memory objects include named entries and regions; they are representations of potentially persistent data that may be mapped into address spaces.
  - **Ports -** Secure, simplex communication channels, accessible only via send and receive capabilities (known as port rights).
  - **IPC -** Message queues, remote procedure calls, notifications, semaphores, and lock sets.

Mach Kernel Abstractions

- These are the main kernel abstractions:
  - **Tasks -** The units of resource ownership; each task consists of a virtual address space, a port right namespace, and one or more threads(Similar to a process.). Allocation of resources to individual threads or groups
  - **Threads** - The units of CPU execution within a task.
  - **Address space -** In conjunction with memory managers, Mach implements the notion of a sparse virtual address space and shared memory.
  - **Memory objects** - The internal units of memory management. Memory objects include named entries and regions; they are representations of potentially persistent data that may be mapped into address spaces.
  - **Ports -** Secure, simplex communication channels, accessible only via send and receive capabilities (known as port rights).
  - **IPC -** Message queues, remote procedure calls, notifications, semaphores, and lock sets.
  - **Time -** Clocks, timers, and waiting.

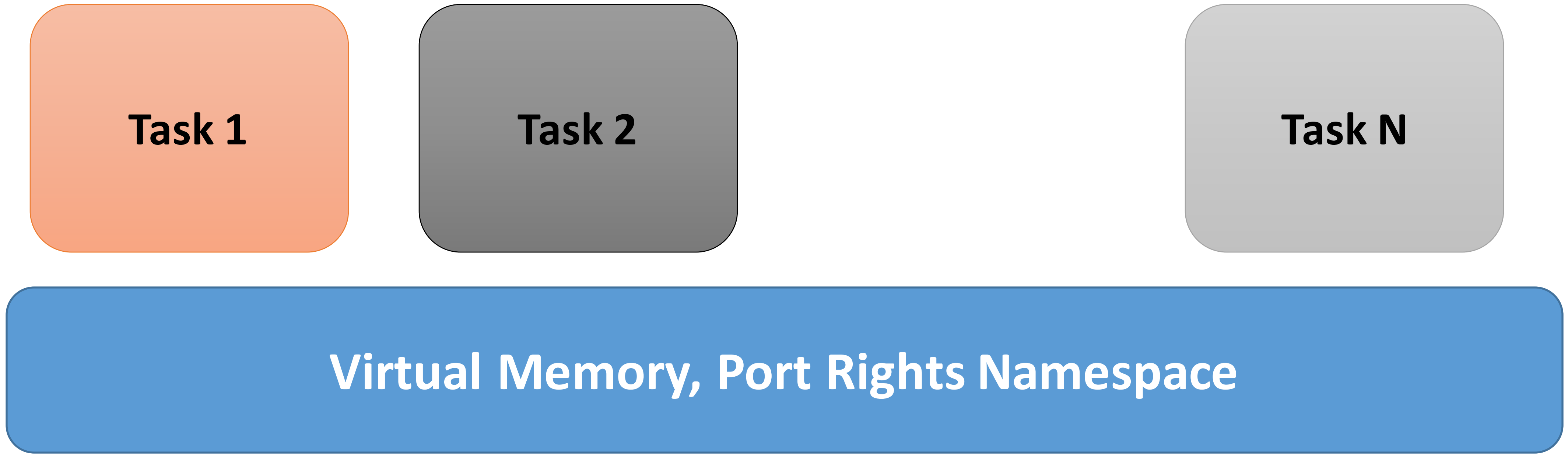
XNU Abstraction



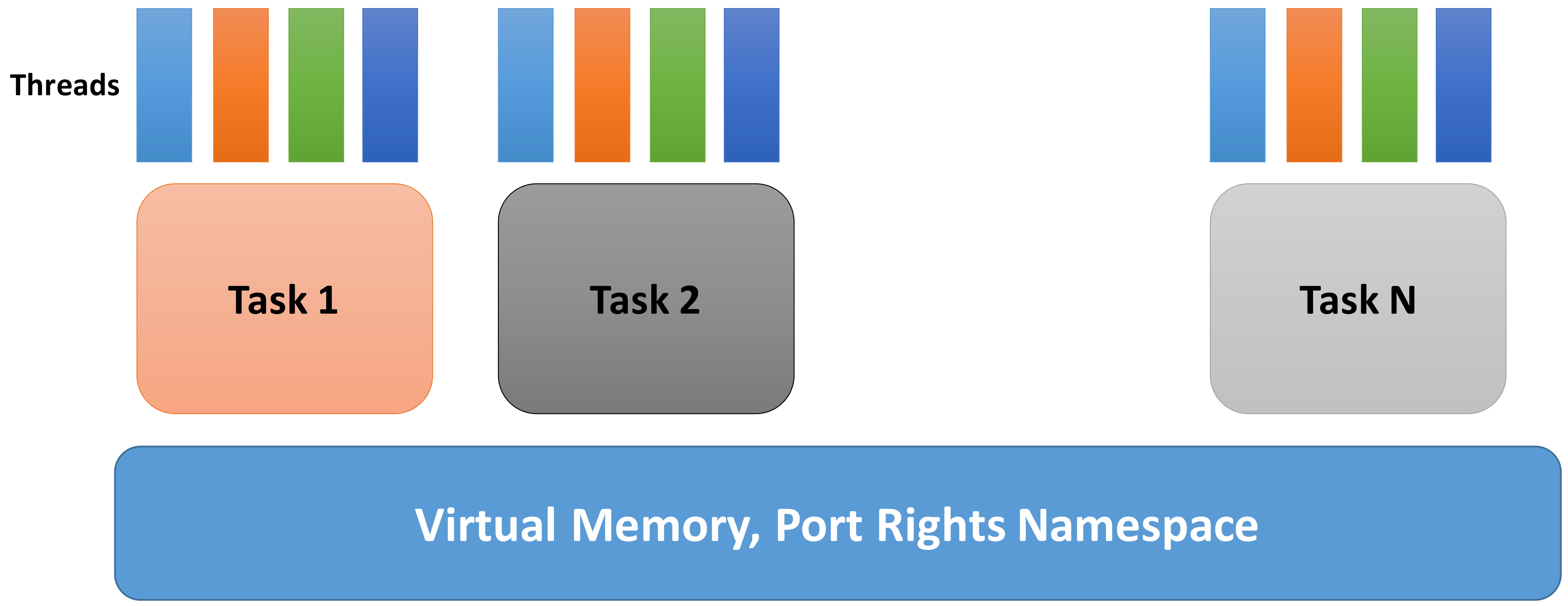
Tasks and Threads

Virtual Memory, Port Rights Namespace

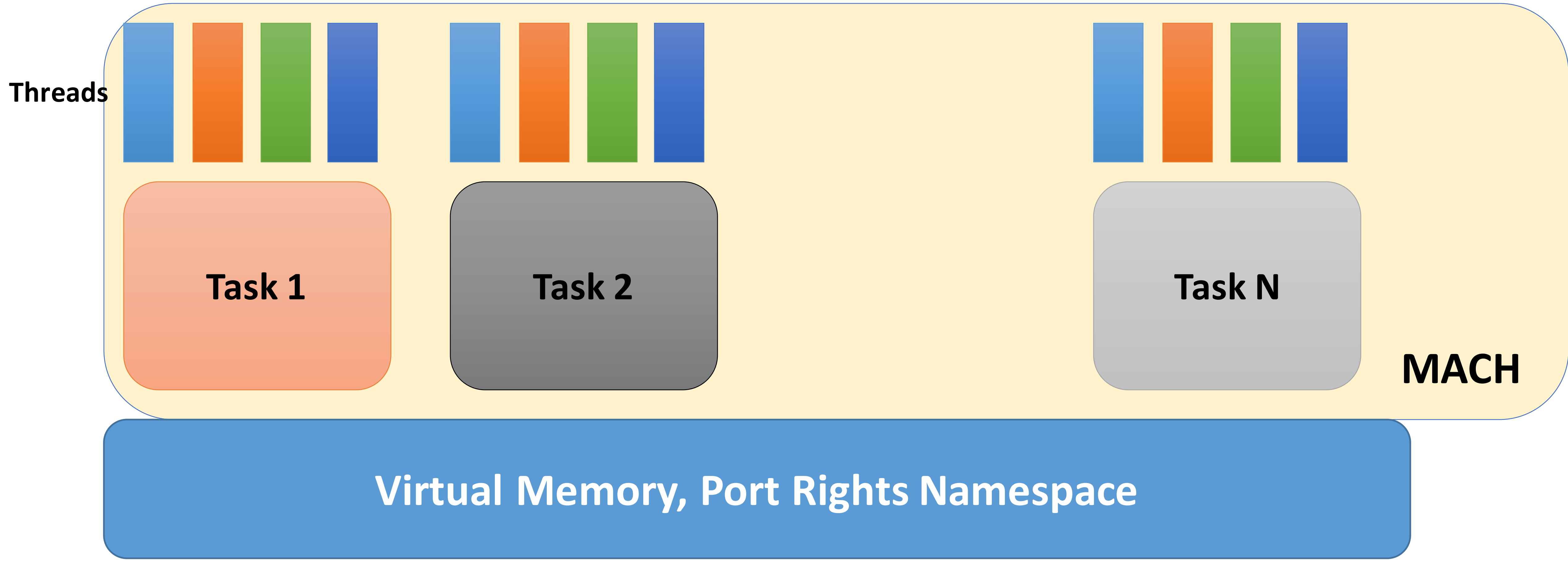
Tasks and Threads



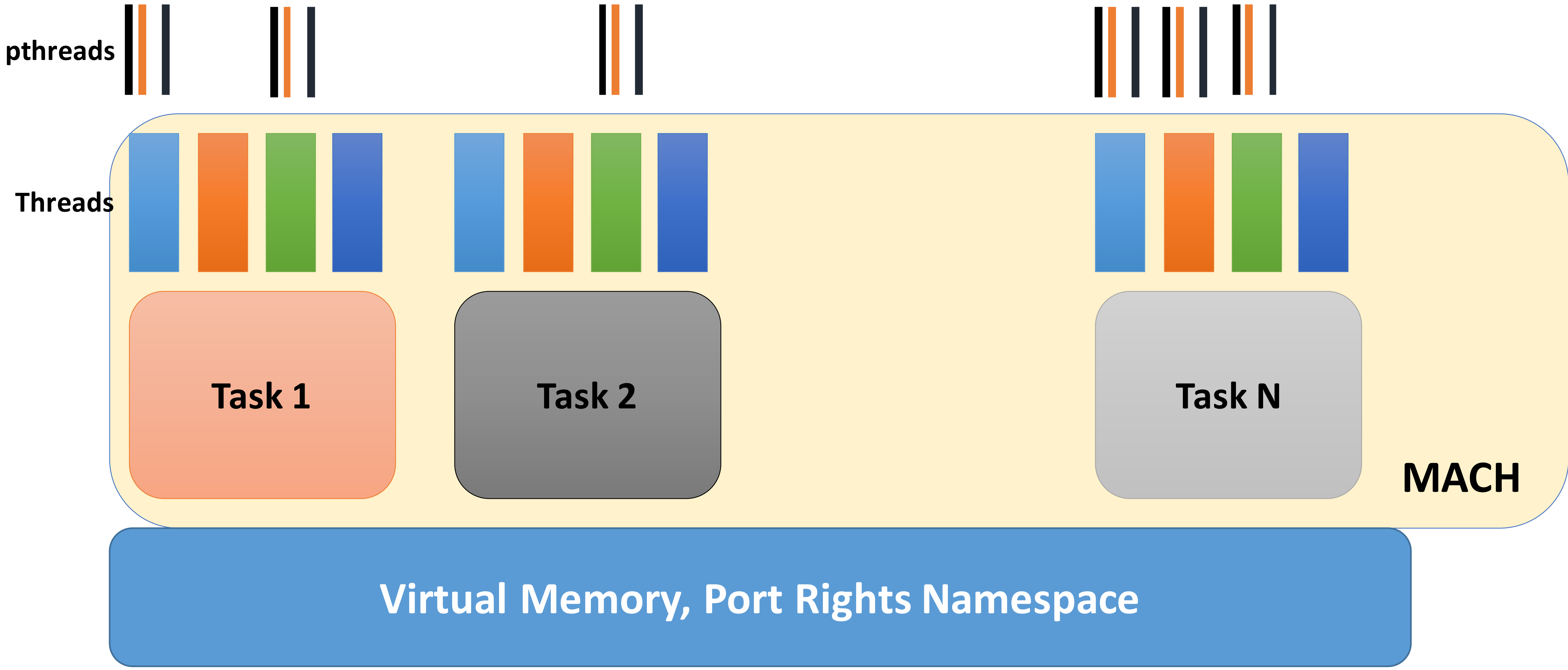
Tasks and Threads



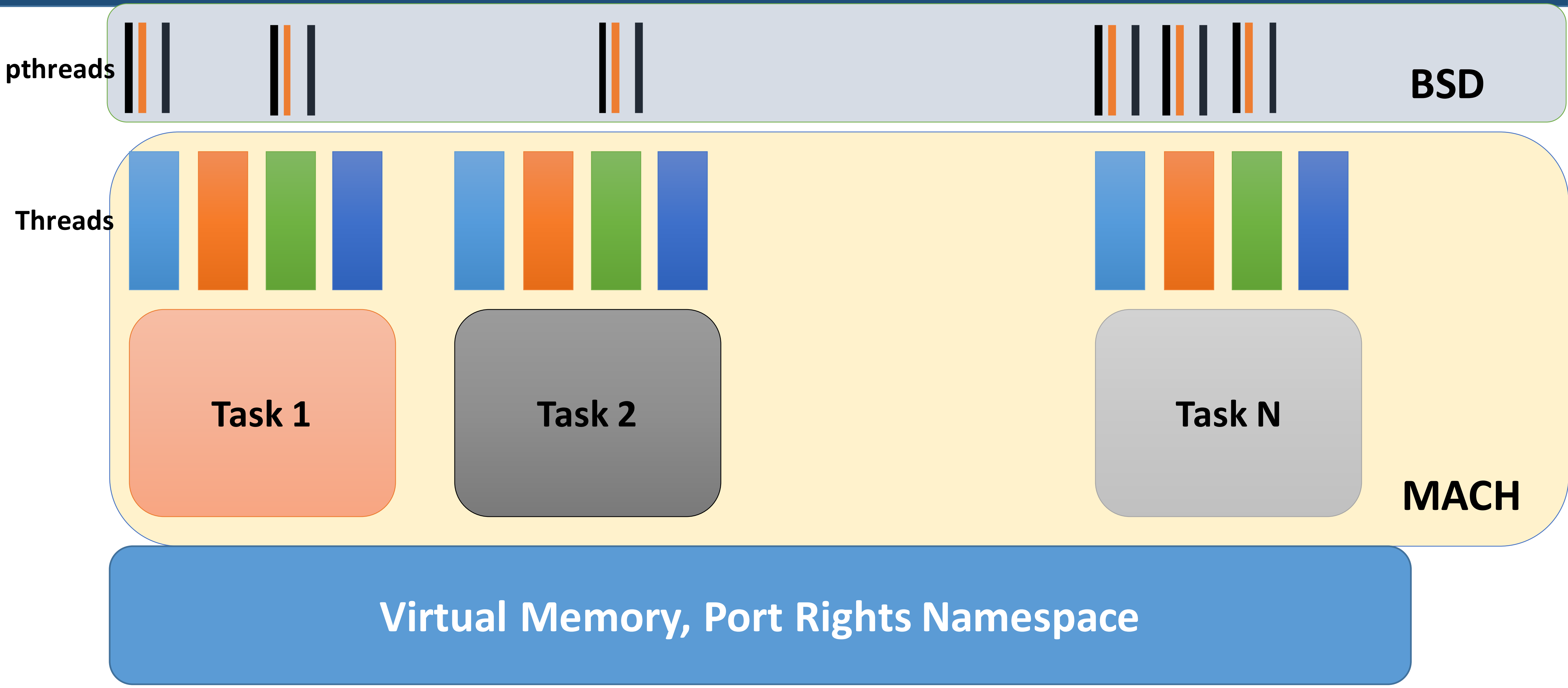
Tasks and Threads



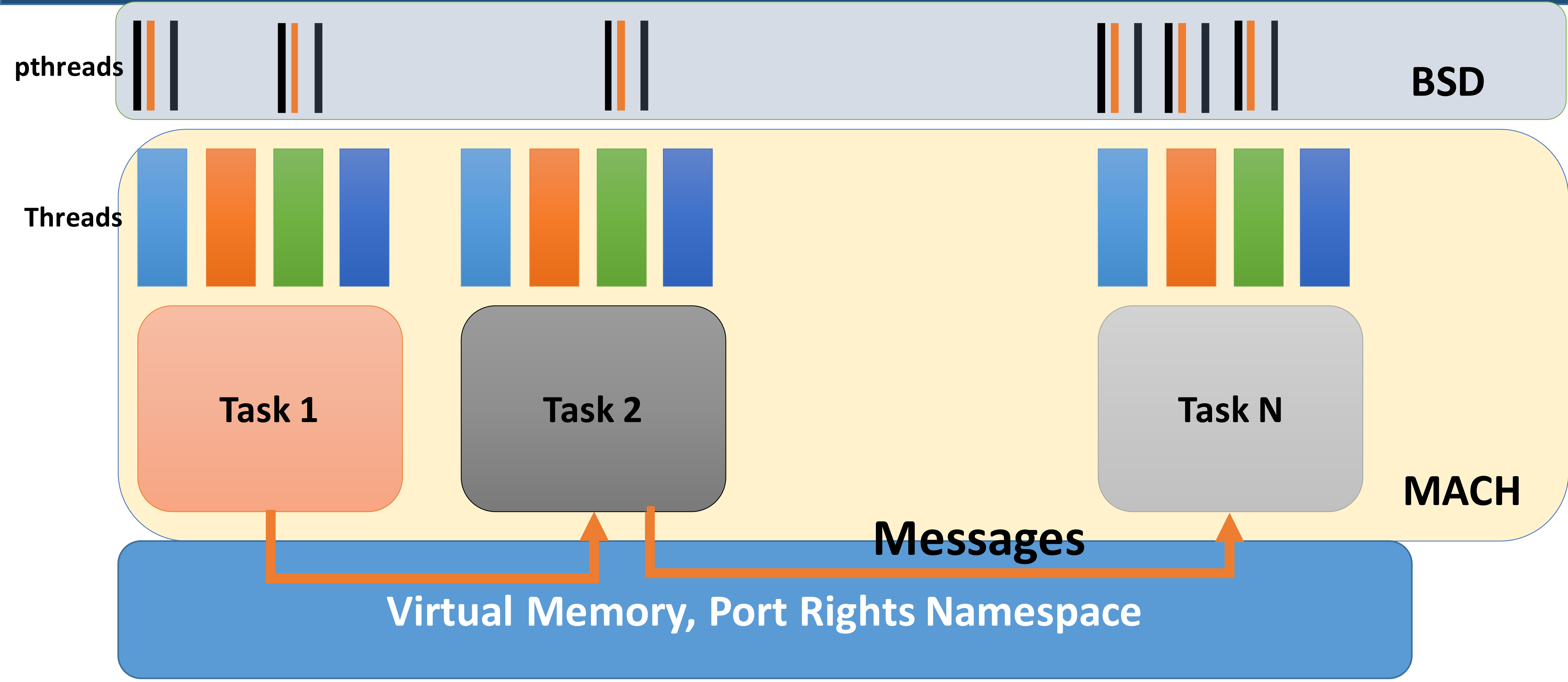
Tasks and Threads



Tasks and Threads



Messages


pthreads
BSD
Threads
Task 1
Task 2
Task N
MACH
Messages
Virtual Memory, Port Rights Namespace

SCHEDULING PRIMITIVES

- It describes tasks and threads, and the application programming interfaces (APIs) they offer.

SCHEDULING PRIMITIVES

- It describes tasks and threads, and the application programming interfaces (APIs) they offer.
- Like all modern operating systems, the kernel sees threads, not processes.

SCHEDULING PRIMITIVES

- It describes tasks and threads, and the application programming interfaces (APIs) they offer.
- Like all modern operating systems, the kernel sees threads, not processes.
- It uses the concepts of the more lightweight tasks rather than processes.

SCHEDULING PRIMITIVES

- It describes tasks and threads, and the application programming interfaces (APIs) they offer.
- Like all modern operating systems, the kernel sees threads, not processes.
- It uses the concepts of the more lightweight tasks rather than processes.
- Classic UNIX uses a top-down approach, in which the basic object is a process that is further divided into one or more threads.

SCHEDULING PRIMITIVES

- It describes tasks and threads, and the application programming interfaces (APIs) they offer.
- Like all modern operating systems, the kernel sees threads, not processes.
- It uses the concepts of the more lightweight tasks rather than processes.
- Classic UNIX uses a top-down approach, in which the basic object is a process that is further divided into one or more threads.
- Mach, on the other hand, uses a bottom-up approach in which the fundamental unit is a thread, and one or more threads are contained in a task.

Threads

- A thread defines the atomic unit of execution in Mach.
- It represents the underlying machine register state and various scheduling statistics.
- Defined in `kern/thread.h` (https://opensource.apple.com/source/xnu/xnu-4570.61.1/osfmk/kern/thread.h.auto.html), a thread is designed to provide the maximum information required for scheduling, while maintaining the lowest overhead possible.
- A thread contains no actual resource references.
- Mach defines the task as a thread container, and it is the task level in which resources are handled.
- A thread has access (via ports) to only the resources and memory allocated in its containing task.

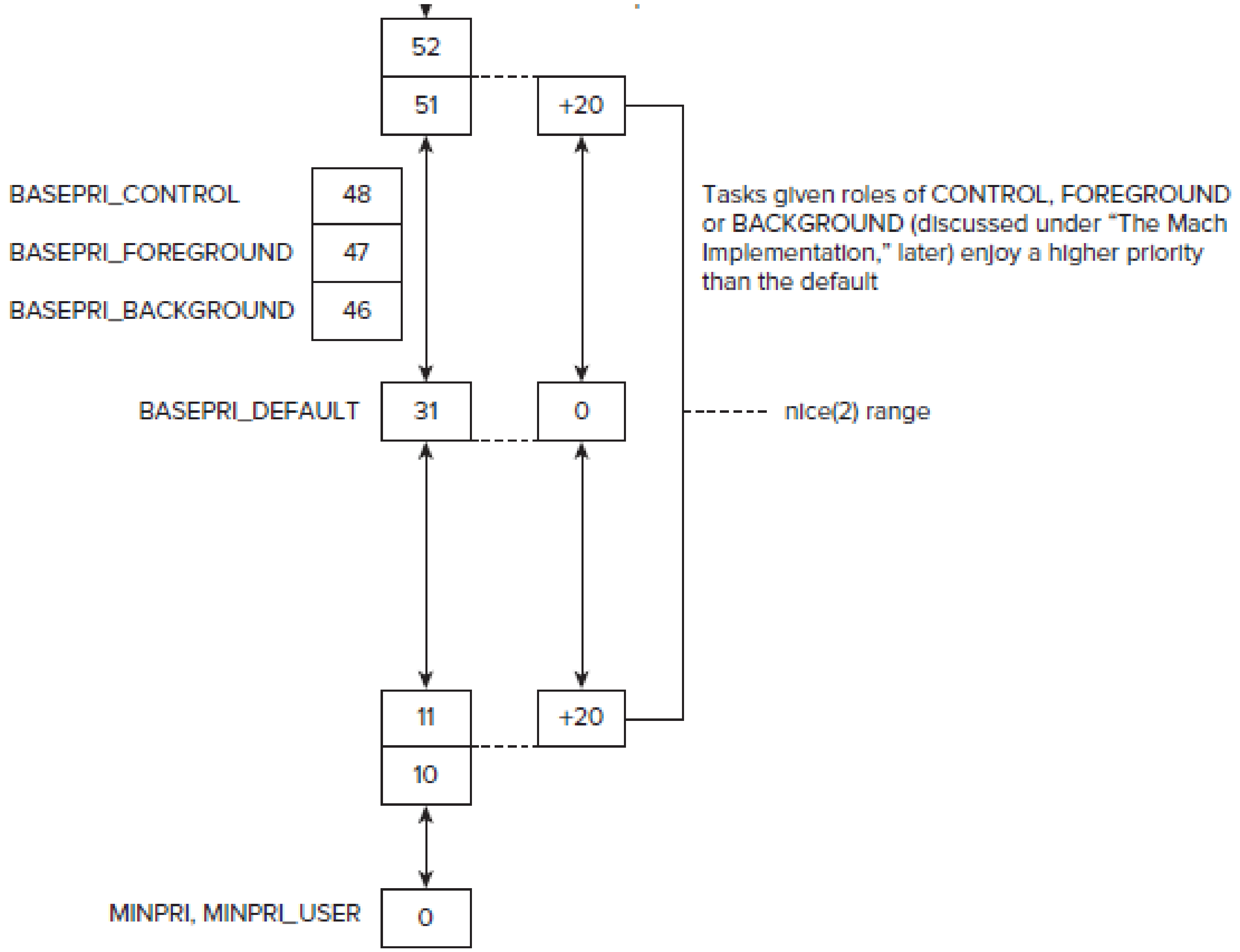
Tasks

- A *task* serves as a container object, under which the virtual memory space and resources are managed.
- These resources are devices and other handles.
- The resources are further abstracted by ports.
- Sharing resources thus becomes a matter of providing access to their corresponding ports.
- The task is a relatively lightweight structure (at least, compared to the threads), defined in `osfmk/kern/task.h` (https://github.com/apple/darwin-xnu/blob/main/osfmk/kern/task.h)

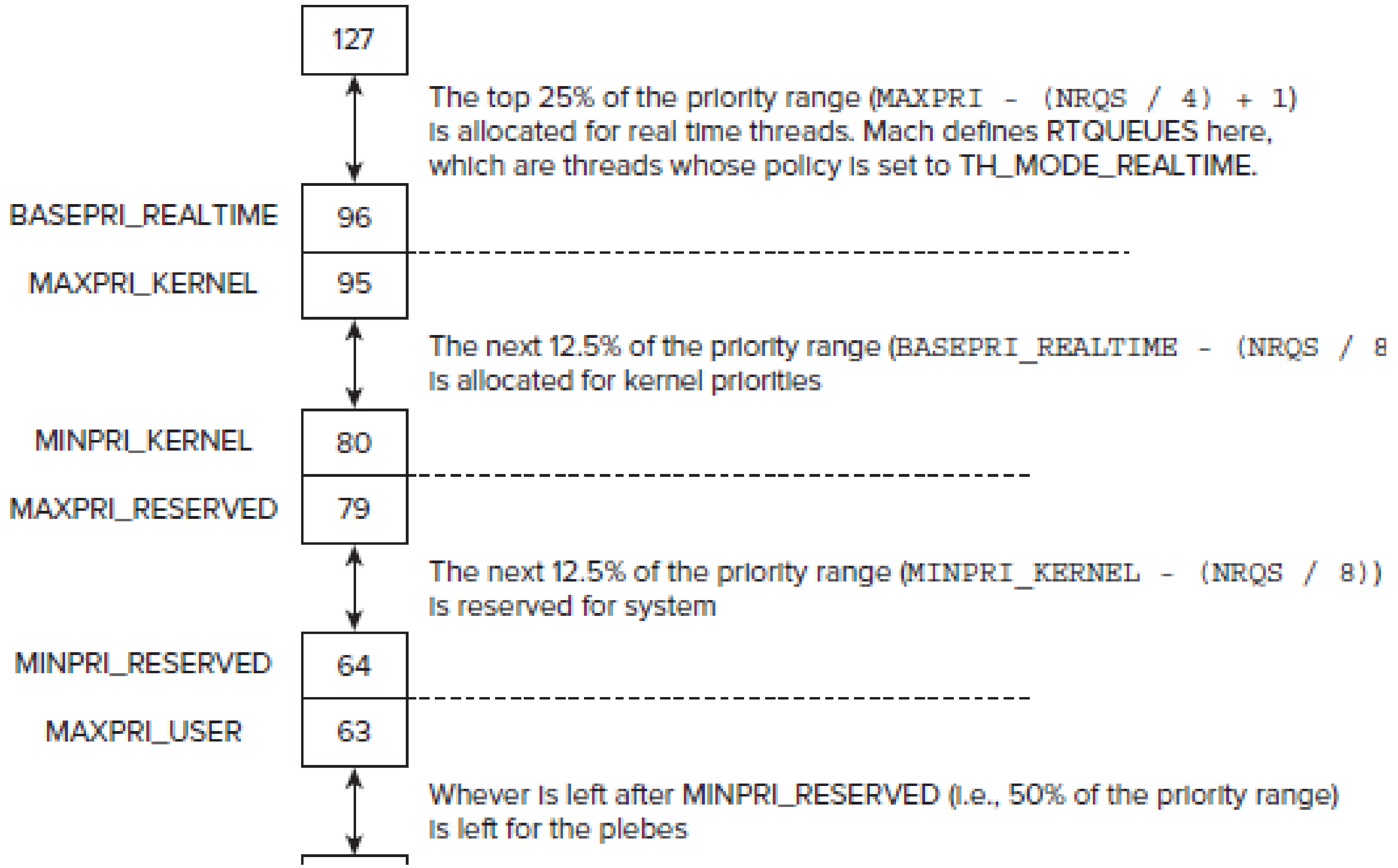
Tasks

- On its own, a task has no life. Its purpose is to serve as a container of one or more threads.
- The threads in a task are maintained in the threads member, which is a queue containing `thread_count` threads.

The Mach Priority Ranges


52
51
+20
BASEPRI_CONTROL
48
BASEPRI_FOREGROUND
47
BASEPRI_BACKGROUND
46
Tasks given roles of CONTROL, FOREGROUND or BACKGROUND (discussed under "The Mach Implementation," later) enjoy a higher priority than the default
BASEPRI_DEFAULT
31
0
nice(2) range
11
+20
10
MINPRI, MINPRI_USER
0

The Mach Priority Ranges



The top 25% of the priority range (`MAXPRI - (NRQS / 4) + 1`) Is allocated for real time threads. Mach defines RTQUEUES here, which are threads whose policy Is set to TH_MODE_REALTIME.

The next 12.5% of the priority range (`BASEPRI_REALTIME - (NRQS / 8` Is allocated for kernel priorities

The next 12.5% of the priority range (`MINPRI_KERNEL - (NRQS / 8))` Is reserved for system

Whever Is left after MINPRI_RESERVED (I.e., 50% of the priority range) Is left for the plebes

Indian Institute of Technology Kharagpur

Priority Shifts

- Mach dynamically tweaks the priorities of each thread during runtime, to accommodate for the thread’s CPU usage, and overall system load.

- Threads can thus “drift” in their priority bands, decreasing in priority when using the CPU too much, and increasing in priority if not getting enough CPU.

Priority Shifts

- The traditional scheduler uses a function (`update_priority`) to update dynamically the priority of each thread.
  - Defined in osfmk/kern/priority.c (https://opensource.apple.com/source/xnu/xnu-7195.81.3/osfmk/kern/priority.c.auto.html)

- The macro toggles the thread priority by subtracting its calculated `sched_usage` (calculated by the function, accounting for CPU usage delta), shifted by a `pri_shift` value.

- The `pri_shift` value is derived from the global `sched_pri_shift`, which is updated by the scheduler regularly as part of the system load calculation in `compute_averages` (osfmk/kern/sched_average.c).

Priority Shifts

- Subtracting the CPU usage delta effectively penalizes those threads with high CPU usage (positive usage delta detracts from priority) and rewards those of low CPU usage (negative usage delta adds to priority).

- To make sure the thread's CPU usage doesn't accrue to the point where the penalty is lethal, the `update_priority` function gradually ages CPU usage.

- It makes use of a `sched_decay_shifts` structure, to simulate the exponential decay of the CPU usage by a factor of (5/8)*n, defined in the `osfmk/kern/priority.c`

- By using the pre-computed shift values, the computation can be speed up, expressed in terms of bit shifts and additions, which take less time than multiplication.

The `sched_decay_shifts` structure in `osfmk/kern/priority.c`

```
/*
* Define shifts for simulating (5/8) ** n
*
* Shift structures for holding update shifts. Actual computation
* is usage = (usage >> shift1) +/- (usage >> abs(shift2)) where the
* +/- is determined by the sign of shift 2.
*/
struct shift_data {
       int shift1;
       int shift2;
};
// The shift data at index i provides the approximation of (5/8)i
#define SCHED_DECAY_TICKS 32
static struct shift_data sched_decay_shifts[SCHED_DECAY_TICKS] = {
{1,1},{1,3},{1,-3},{2,-7},{3,5},{3,-5},{4,-8},{5,7},
{5,-7},{6,-10},{7,10},{7,-9},{8,-11},{9,12},{9,-11},{10,-13},
{11,14},{11,-13},{12,-15},{13,17},{13,-15},{14,-17},{15,19},{16,18},
{16,-19},{17,22},{18,20},{18,-20},{19,26},{20,22},{20,-22},{21,-27}
};
```

Scheduling Algorithms

- Mach’s thread scheduling is highly extensible, and actually allows changing the algorithms used for thread scheduling (osfmk/kern/sched_prim.h)

Supported schedulers in Mach	
KSCHED… CONSTANT (STRING)	**USED FOR**
SCHED_TRADITIONAL	Traditional (default)
SCHED_PROTO	Global runqueue based scheduler
SCHED_GRRR	Group Ratio Round Robin
SCHED_MULTIQ	Traditional multi-queue ready queue scheduler
SCHED_CLUTCH	Schedule group of threads
SCHED_EDGE	Better control over various QoS buckets

SCHED_CLUTCH - Schedule group of threads

- The clutch scheduler schedules groups of threads instead of individual threads.

- Breaking away from the traditional single-tier scheduling model, it implements a hierarchical scheduler which makes optimal decisions at various thread grouping levels.

- The hierarchical scheduler currently has 3 levels:
  - Scheduling Bucket Level
  - Thread Group Level
  - Thread Level

SCHED_CLUTCH - Schedule group of threads

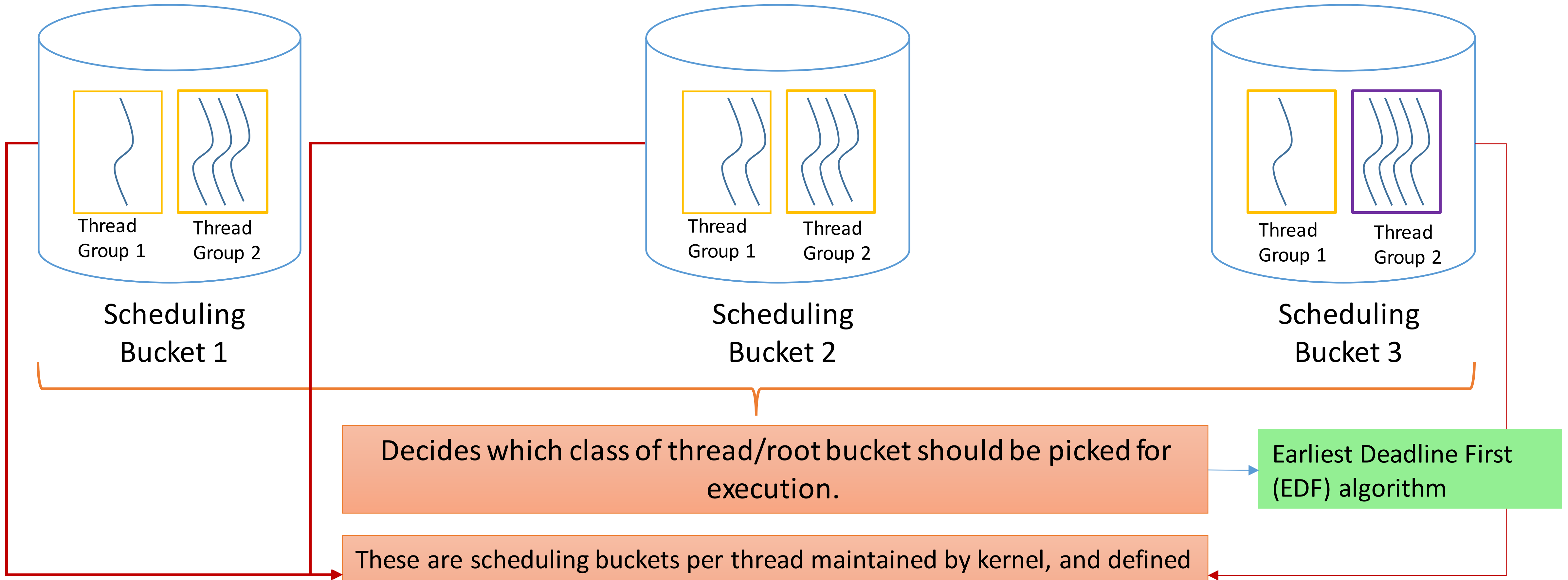
- Clutch ordering based on thread group flags (specified by the thread grouping mechanism).
- These properties define a thread group specific priority boost.
- The current implementation gives a slight boost to HIGH & MED thread groups which effectively deprioritizes daemon thread groups which are marked "Efficient" on AMP systems.
- Bound threads are not managed in the clutch hierarchy.
- How to indicate if the thread should be in the hierarchy or not?

SCHED_CLUTCH - Schedule group of threads

- In the clutch scheduler, the threads are maintained in runqs at the clutch_bucket level (clutch_bucket defines a unique thread group and scheduling bucket pair). The thread is linked via a couple of linkages in the clutch bucket:
  - A stable priority queue linkage which is the main runqueue (based on sched_pri) for the clutch bucket
  - A regular priority queue linkage which is based on thread's base/promoted pri (used for clutch bucket priority calculation)
  - A queue linkage used for timesharing operations of threads at the scheduler tick

- Since the clutch scheduler organizes threads based on the thread group and the scheduling bucket, its important to not mix threads from multiple priority bands into the same bucket. To achieve that, in the clutch bucket world, there is a scheduling bucket per QoS effectively.

SCHED_CLUTCH - Scheduling Bucket Level



Decides which class of thread/root bucket should be picked for execution.

Earliest Deadline First (EDF) algorithm

These are scheduling buckets per thread maintained by kernel, and defined based on the base/scheduling priority of the threads.

All runnable threads with the same scheduling bucket are represented by a single entry in a priority queue which is ordered by the bucket's deadline.

SCHED_CLUTCH - Scheduling Bucket Level

- These entries are known as **root buckets** throughout the implementation.
- The goal of this level is to provide low latency access to the CPU for high QoS classes while ensuring starvation avoidance for the low QoS classes.
- The bucket selection algorithm simply selects the root bucket with the earliest deadline in the priority queue.
- The deadline for a root bucket is calculated based on its first-runnable timestamp and its **Worst Case Execution Latency (WCEL)** value which is pre-defined for each bucket.
- The WCEL values are picked based on the decay curve followed by the Mach timesharing algorithm to allow the system to function similar to the existing scheduler from a higher level perspective.

SCHED_CLUTCH - Scheduling Bucket Level

```
static uint32_t sched_clutch_root_bucket_wcel_us[TH_BUCKET_SCHED_MAX] = {
        SCHED_CLUTCH_INVALID_TIME_32,                   /* FIXPRI */
        0,                                              /* FG */
        37500,                                          /* IN (37.5ms) */
        75000,                                          /* DF (75ms) */
        150000,                                         /* UT (150ms) */
        250000                                          /* BG (250ms) */
};
```

SCHED_CLUTCH - Scheduling Bucket Level

- Whenever a root bucket transitions from non-runnable to runnable, its deadline is set to (now + WCEL[bucket]).

- This ensures that the bucket would be scheduled at WCEL[bucket] even in a heavily loaded system.

- Once the root bucket is picked for execution, its deadline is pushed by WCEL[bucket] into the future.

- For better performance, the bucket level scheduler implements a **root bucket warp mechanism.** Each bucket is provided a warp value which is refreshed whenever the bucket is selected due to its deadline expiring.

SCHED_CLUTCH - Scheduling Bucket Level

```
static uint32_t sched_clutch_root_bucket_warp_us[TH_BUCKET_SCHED_MAX] = {
        SCHED_CLUTCH_INVALID_TIME_32,                       /* FIXPRI */
        8000,                                               /* FG (8ms)*/
        4000,                                               /* IN (4ms) */
        2000,                                               /* DF (2ms) */
        1000,                                               /* UT (1ms) */
        0                                                   /* BG (0ms) */
};
```

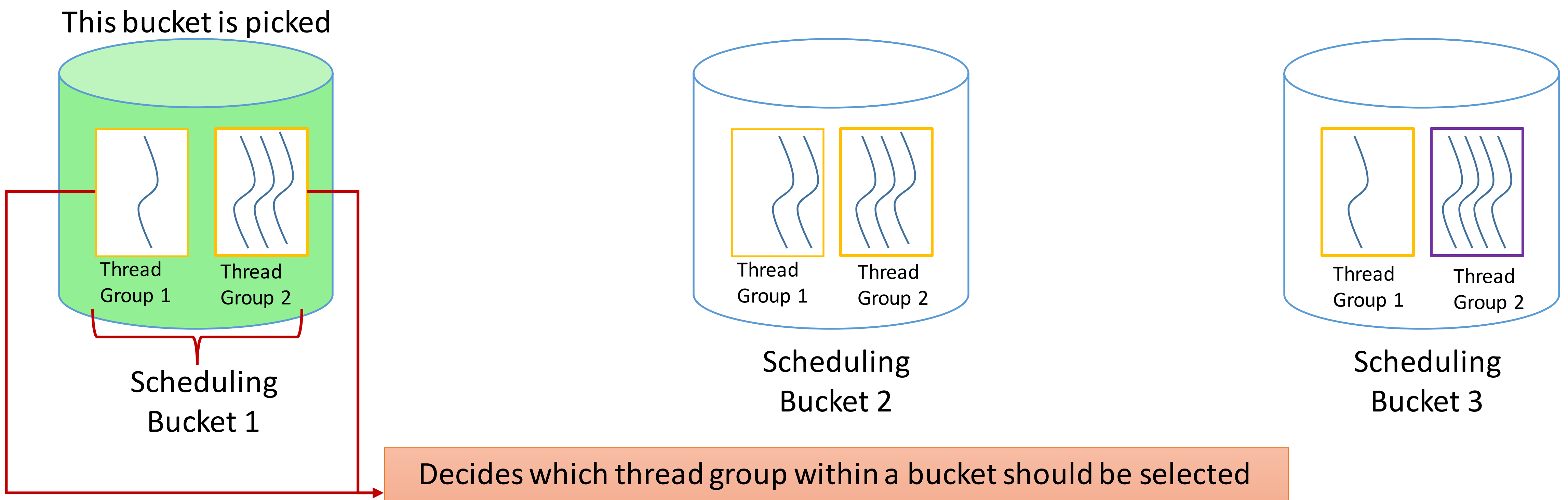
SCHED_CLUTCH - Scheduling Bucket Level

- The root bucket selection logic finds the earliest deadline bucket and then checks if there are any higher (in natural priority order) buckets that have warp remaining.
- If there is such a higher bucket, it would select that bucket and effectively open a warp window.
- During this warp window the scheduler would continue to select this warping bucket over lower priority buckets. Once the warping bucket is drained or the warp window expires, the scheduler goes back to scheduling buckets in deadline order.
- This mechanism provides a bounded advantage to higher level buckets to allow them to remain responsive in the presence of bursty workloads.
- The scheduling bucket level also maintains a bitmap of runnable root buckets to allow quick checks for empty hierarchy and root level priority calculation.

Earliest Deadline First (EDF) algorithm

- The EDF algorithm is the best choice for this level due to the following reasons:
  - Deadline based scheduling allows the scheduler to define strict bounds on worst case execution latencies for all scheduling buckets.
  - The EDF algorithm is dynamic based on bucket runnability and selection. Since all deadline updates are computationally cheap, the algorithm can maintain up-to-date information without measurable overhead.
  - It achieves the goals of maintaining low scheduling latency for high buckets and starvation avoidance for low buckets efficiently.
  - Since the bucket level scheduler deals with a fixed small number of runnable buckets in the worst case, it is easy to configure in terms of defining deadlines, warps etc.

SCHED_CLUTCH - Thread Group Level



Decides which thread group within a bucket should be selected next for execution.

Thread groups are a collection of threads working on behalf of a specific workload.

Each thread group with runnable threads within a bucket is represented as an entry at this level.

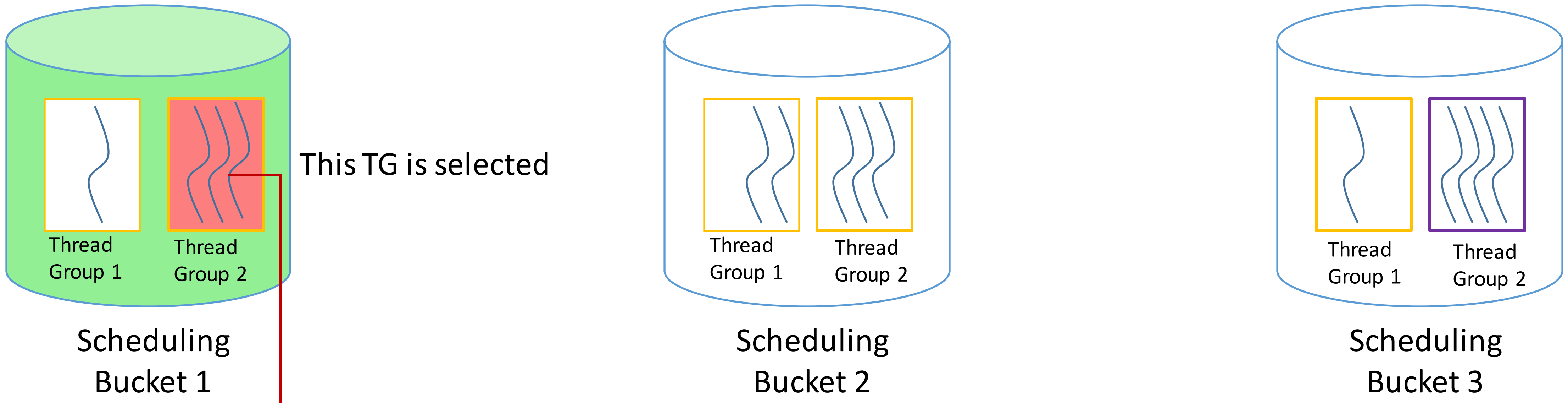
SCHED_CLUTCH - Thread Group Level

- These entries are known as **clutch buckets** throughout the implementation.
- The goal of this level is to share the CPU among various user workloads with preference to interactive applications over compute-intensive batch workloads.
- The clutch bucket selection algorithm simply selects the clutch bucket with the highest priority in the priority queue.
- The priority calculation for the clutch buckets is based on the following factors:
  - **Highest runnable thread in the clutch bucket**:
  - **Interactivity score**
  - **Thread Group Type**

SCHED_CLUTCH - Interactivity score based algorithm

- The interactivity score based algorithm is well suited for this level due to the following reasons:
    - It allows for a fair sharing of CPU among thread groups based on their recent behavior. Since the algorithm only looks at recent CPU usage history, it also adapts to changing behavior quickly.
    - Since the priority calculation is fairly cheap, the scheduler is able to maintain up-to-date information about all thread groups which leads to more optimal decisions.
    - Thread groups provide a convenient abstraction for groups of threads working together for a user workload. Basing scheduling decisions on this abstraction allows the system to make interesting choices such as preferring Apps over daemons which is typically better for system responsiveness.

SCHED_CLUTCH - Thread Level

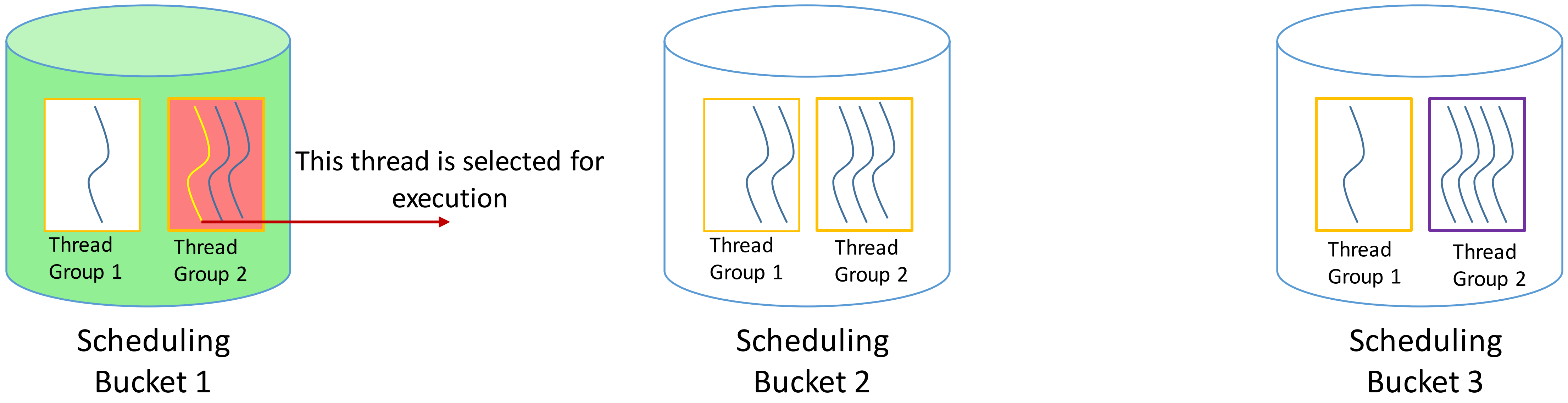


Decides which thread within a clutch bucket should be selected next for execution.

Each runnable thread in the clutch bucket is represented as an entry in a runqueue which is organized based on the schedpri of threads.

The thread selection algorithm simply selects the highest priority thread in the runqueue.

SCHED_CLUTCH - Thread Level



The schedpri calculation for the threads is based on the traditional Mach scheduling algorithm which uses load & CPU usage to decay priority for a thread.

The thread decay model is more suited at this level as compared to the global scheduler because the load calculation only accounts for threads in the same clutch bucket.

Since all threads in the same clutch bucket belong to the same thread group and scheduling bucket, this algorithm provides quick CPU access for latency sensitive threads within the clutch bucket without impacting other non-related threads in the system.

Scheduler Priority Calculations - Thread Level Implementation

- The thread level scheduler implements the Mach timesharing algorithm to decide which thread within the clutch bucket should be selected next for execution.
- All runnable threads in a clutch bucket are inserted into the runqueue based on the **schedpri**.
- The scheduler calculates the **schedpri** of the threads in a clutch bucket based on the number of runnable threads in the clutch bucket and the CPU usage of individual threads.

Scheduler Priority Calculations - Thread Level Implementation

- The load information is updated every scheduler tick and the threads use this information for priority decay calculation as they use CPU.

- **The priority decay algorithm** attempts to reward bursty interactive threads and penalize CPU intensive threads. Once a thread is selected for running, it is assigned a quantum which is based on the scheduling bucket it belongs to.

- The quanta for various buckets are defined statically as:

```
static uint32_t sched_clutch_thread_quantum_us[TH_BUCKET_SCHED_MAX] = {
    10000, /* FIXPRI (10ms) */
    10000, /* FG (10ms) */
    8000, /* IN (8ms) */
    6000, /* DF (6ms) */
    4000, /* UT (4ms) */
    2000 /* BG (2ms) */ };
```

SCHED_CLUTCH - Schedule group of threads

- The clutch scheduler organizes the threads based on the thread group and the scheduling bucket. The Buckets are :

```
    TH_BUCKET_FIXPRI = 0,    /* Fixed-priority */
    TH_BUCKET_SHARE_FG,    /* Timeshare thread above BASEPRI_DEFAULT */
    /* Timeshare thread between BASEPRI_USER_INITIATED and BASEPRI_DEFAULT 
*/
    TH_BUCKET_SHARE_IN
     /* Timeshare thread between BASEPRI_DEFAULT and BASEPRI_UTILITY */
    TH_BUCKET_SHARE_DF
    /* Timeshare thread between BASEPRI_UTILITY and MAXPRI_THROTTLE */
    TH_BUCKET_SHARE_UT
     /* Timeshare thread between MAXPRI_THROTTLE and MINPRI */
    TH_BUCKET_SHARE_BG
    TH_BUCKET_RUN,                          /* All runnable threads */
    TH_BUCKET_SCHED_MAX = TH_BUCKET_RUN,    /* Maximum schedulable buckets 
*/
    TH_BUCKET_MAX,
```

Scheduler Priority Calculations - Root Priority Calculation

- The scheduler maintains a root level priority for the hierarchy in order to make decisions regarding pre-emptions and thread selection.
- The root priority is updated as threads are inserted/removed from the hierarchy.
- The root level also maintains the urgency bits to help with preemption decisions.
- Since the root level priority/urgency is used for preemption decisions, it is based on the threads in the hierarchy and is calculated as :

Scheduler Priority Calculations - Root Priority Calculation

- Root Priority Calculation:

  1. If AboveUI bucket is runnable,

  2. Compare priority of AboveUI highest clutch bucket (CBUI) with Timeshare FG highest clutch bucket (CBFG)

  3. If pri(CBUI) >= pri(CBFG), select CBUI

  4. Otherwise find the (non-AboveUI) highest priority root bucket that is runnable and select its highest clutch bucket

  5. Find the highest priority (promoted or base pri) thread within that clutch bucket and assign that as root priority

Scheduler Priority Calculations - Root Priority Calculation

- Root Priority Calculation:

  1. If AboveUI bucket is runnable,
  2. Compare priority of AboveUI highest clutch bucket (CBUI) with Timeshare FG highest clutch bucket (CBFG)
  3. If pri(CBUI) >= pri(CBFG), select CBUI
  4. Otherwise find the (non-AboveUI) highest priority root bucket that is runnable and select its highest clutch bucket
  5. Find the highest priority (promoted or base pri) thread within that clutch bucket and assign that as root priority

- Root Urgency Calculation:

  1. On thread insertion into the hierarchy, increment the root level urgency based on thread's sched_pri
  2. On thread removal from the hierarchy, decrement the root level urgency based on thread's sched_pri

Scheduler Priority Calculations - Root Bucket Priority Calculation

- The root bucket priority is simply the deadline of the root bucket which is calculated by adding the WCEL of the bucket to the timestamp of the root bucket becoming runnable.

root-bucket priority = now + WCEL[bucket]

Scheduler Priority Calculations - Clutch Bucket Priority Calculation

- As mentioned earlier, the priority value of a clutch bucket is calculated based on the highest runnable thread, interactivity score and the thread group type.

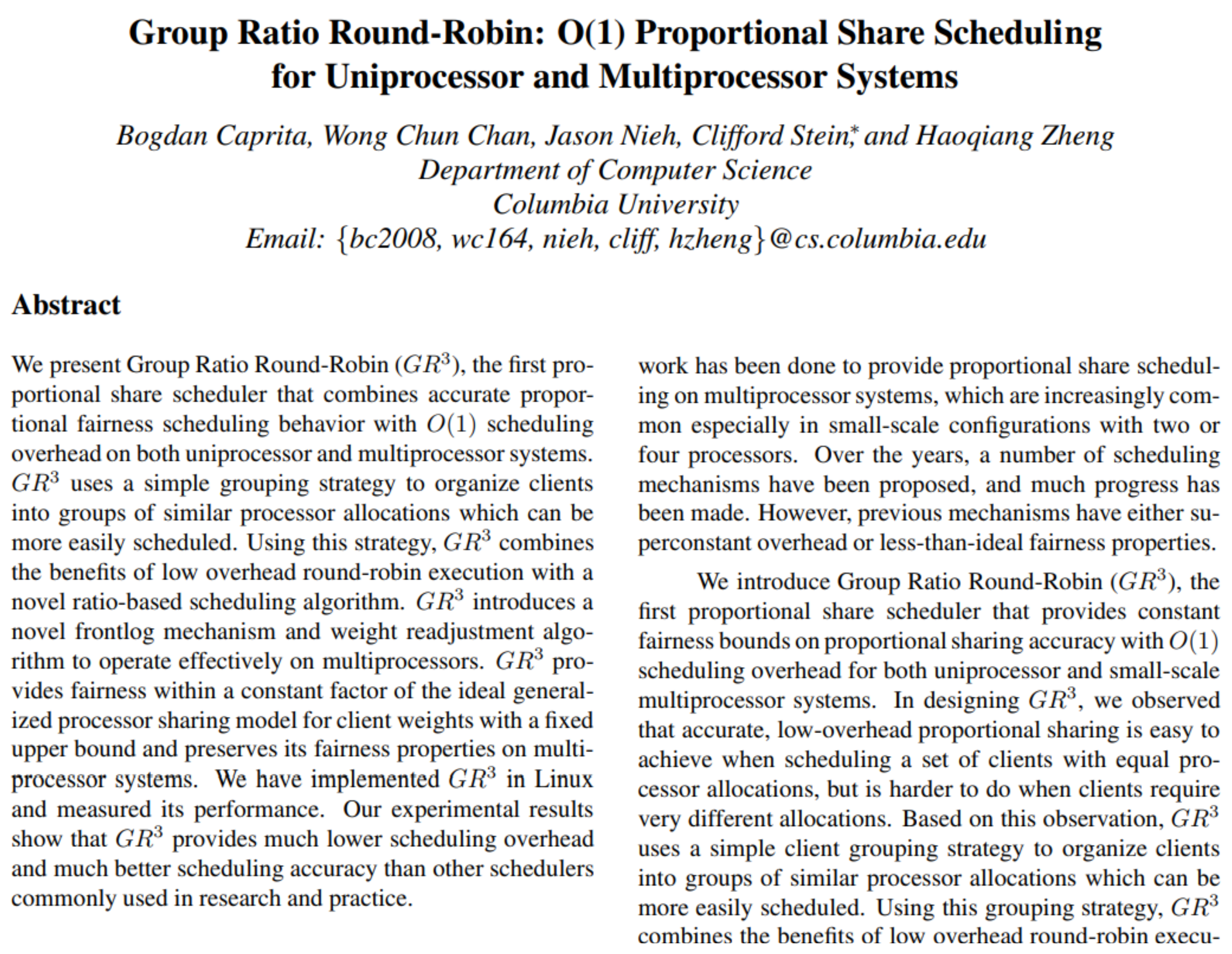
1. Find the highest runnable thread (promoted or basepri) in the clutch bucket (maxpri)
2. Check if the thread group for this clutch bucket is marked Efficient.
3. If not, assign a positive boost value (clutch_boost)
4. Calculate the ratio of CPU blocked and CPU used for the clutch bucket.
5. If blocked > used, assign a score (interactivity_score) in the higher range.
6. Else, assign a score (interactivity_score) in the lower range.
7. clutch-bucket priority = maxpri + clutch_boost + interactivity_score

Scheduler Priority Calculations - Thread Priority Calculation

- The thread priority calculation is based on the Mach timesharing algorithm.

1. At every scheduler tick, snapshot the load for the clutch bucket
2. Use the load value to calculate the priority shift values for all threads in the clutch bucket
3. thread priority = base priority - (thread CPU usage >> priority shift)

SCHED_GRRR – Group Ratio Round Robin Scheduler



Group Ratio Round-Robin: O(1) Proportional Share Scheduling for Uniprocessor and Multiprocessor Systems

Bogdan Caprita, Wong Chun Chan, Jason Nieh, Clifford Stein, and Haoqiang Zheng*
Department of Computer Science
Columbia University
Email: {bc2008, wc164, nieh, cliff, hzheng}@cs.columbia.edu

Abstract

We present Group Ratio Round-Robin (GR^3), the first proportional share scheduler that combines accurate proportional fairness scheduling behavior with $O(1)$ scheduling overhead on both uniprocessor and multiprocessor systems. GR^3 uses a simple grouping strategy to organize clients into groups of similar processor allocations which can be more easily scheduled. Using this strategy, GR^3 combines the benefits of low overhead round-robin execution with a novel ratio-based scheduling algorithm. GR^3 introduces a novel frontlog mechanism and weight readjustment algorithm to operate effectively on multiprocessors. GR^3 provides fairness within a constant factor of the ideal generalized processor sharing model for client weights with a fixed upper bound and preserves its fairness properties on multiprocessor systems. We have implemented GR^3 in Linux and measured its performance. Our experimental results show that GR^3 provides much lower scheduling overhead and much better scheduling accuracy than other schedulers commonly used in research and practice.

work has been done to provide proportional share scheduling on multiprocessor systems, which are increasingly common especially in small-scale configurations with two or four processors. Over the years, a number of scheduling mechanisms have been proposed, and much progress has been made. However, previous mechanisms have either superconstant overhead or less-than-ideal fairness properties.

We introduce Group Ratio Round-Robin (GR^3), the first proportional share scheduler that provides constant fairness bounds on proportional sharing accuracy with $O(1)$ scheduling overhead for both uniprocessor and small-scale multiprocessor systems. In designing GR^3, we observed that accurate, low-overhead proportional sharing is easy to achieve when scheduling a set of clients with equal processor allocations, but is harder to do when clients require very different allocations. Based on this observation, GR^3 uses a simple client grouping strategy to organize clients into groups of similar processor allocations which can be more easily scheduled. Using this grouping strategy, GR^3 combines the benefits of low overhead round-robin execu-

USENIX ATC 2005

SCHED_GRRR - Group Ratio Round Robin

- The Group Ratio Round-Robin (GR^3) scheduler is the first proportional share scheduler [1] that combines accurate proportional fairness scheduling[2] behavior with O(1) scheduling overhead on both uniprocessor and multiprocessor systems, i.e. it provides constant fairness bounds on proportional sharing accuracy with O(1) scheduling overhead.

- GR^3 uses a simple **grouping strategy** to organize clients into groups of similar processor allocations which can be more easily scheduled.
- Why to use grouping strategy?
  - Observations: Accurate, low-overhead proportional sharing is easy to achieve when scheduling a set of clients with equal processor allocations, but is harder to do when clients require very different allocations

[1] In a proportional share algorithm every job has a weight, and jobs receive a share of the available resources proportional to the weight of every job.
[2] Proportional-fair scheduling is a compromise-based scheduling algorithm. It is based upon maintaining a balance between two competing interests.

SCHED_GRRR - Group Ratio Round Robin

- Using this strategy, GR3 combines the benefits of low overhead round-robin execution with a novel ratio-based scheduling algorithm

- GR3 introduces a novel **frontlog mechanism** and **weight readjustment algorithm** to operate effectively on multiprocessors.

- Why to use frontlog mechanism?
  - Unlike uniprocessor system, on a multiprocessor system, a client may not be able to be scheduled to run on a processor because it is currently running on another processor.
  - To preserve its fairness properties, GR^3 keeps track of a **frontlog** per client to indicate when the client was already running but could have been scheduled to run on another processor (based on cache affinity).

SCHED_GRRR - Group Ratio Round Robin

- It then assigns the client a time quantum that is added to its allocation on the processor it is running on.

- The frontlog ensures that a client receives its proportional share allocation while also taking advantage of any cache affinity by continuing to run the client on the same processor.

https://www.usenix.org/legacy/events/usenix05/tech/general/full_papers/caprita/caprita.pdf

GRRR – Weight Readjustment Algorithm

- The weight readjustment algorithm also takes advantage of its grouping strategy.
- On a multiprocessor system, proportional sharing is not feasible for some client weight assignments, such as having one client with weight 1 and another with weight 2 on a two-processor system.
- By organizing clients with similar weights into groups, GR^3 adjusts for infeasible weight assignments without the need to order clients, resulting in lower scheduling complexity than previous approaches.

https://www.usenix.org/legacy/events/usenix05/tech/general/full_papers/caprita/caprita.pdf

GRRR – Uniprocessor Scheduling

Uniprocessor scheduling, the process of scheduling a time multiplexed resource among a set of clients, has two basic steps:

1. order the clients in a queue
2. run the first client in the queue for its time quantum, which is the maximum time interval the client is allowed to run before another scheduling decision is made

https://www.usenix.org/legacy/events/usenix05/tech/general/full_papers/caprita/caprita.pdf

GRRR – Uniprocessor Scheduling

- A scheduler can achieve proportional sharing in one of two ways.

1. **Fair queueing** - to adjust the frequency that a client is selected to run by adjusting the position of the client in the queue so that it ends up at the front of the queue more or less often.

Cons - adjusting the client's position in the queue typically requires sorting clients based on some metric of fairness and has a time complexity that grows with the number of clients.

GRRR – Uniprocessor Scheduling

2. Adjust the size of a client's time quantum so that it runs longer for a given allocation, as is done in **weighted round-robin** (WRR)

Pros - Fast, providing constant time complexity scheduling overhead.

Cons - Allowing a client to monopolize the resource for a long period of time that results in an extended periods of unfairness to other clients which receive no service during those times. The unfairness is worse with skewed weight distributions.

GRRR – The High level view

At a high-level, the GRRR scheduling algorithm can be briefly described in three parts:

1. Client grouping strategy
2. Intergroup scheduling
3. Intragroup scheduling

GRRR – Notations for clients

- For each client, GR3 maintains the following three values:
  - Weight
  - Deficit, and
  - Run state

- Each client receives a resource allocation that is directly proportional to its ***weight***.

- A client’s ***deficit*** tracks the number of remaining time quanta the client has not received from previous allocations.

- A client’s ***run state*** indicates whether or not it can be executed. A client is ***runnable*** if it can be executed.

GRRR – Notations for groups

- For each group, GR3 maintains the following four values:
  - group weight, group order, group work, and current client.

- The ***group weight*** is the sum of the corresponding weights of the clients in the group run queue.

- A group with ***group order k*** contains the clients with weights between 2^k to $2^{k+1} - 1$.

- The ***group work*** is the total execution time clients in the group have received.

- The ***current client*** is the most recently scheduled client in the group's run queue.

GRRR – Notations for scheduler state

- GR3 also maintains the following scheduler state:
  - time quantum, group list, total weight, and current group.

- The ***group list*** is a sorted list of all groups containing runnable clients ordered from largest to smallest group weight, with ties broken by group order.

- The ***total weight*** is the sum of the weights of all runnable clients.

- The ***current group*** is the most recently selected group in the group list.

GRRR – Client grouping strategy

- Clients are separated into groups of clients with similar weight values.

- The group of order k is assigned all clients with weights between 2^k to $2^{k+1} - 1$, where $k \geq 0$.

GRRR – Intergroup scheduling

- Groups are ordered in a list from largest to smallest group weight, where the group weight of a group is the sum of the weights of all clients in the group.
- Groups are selected in a round robin manner based on the ratio of their group weights.
- If a group has already been selected more than its proportional share of the time, move on to the next group in the list. Otherwise, skip the remaining groups in the group list and start selecting groups from the beginning of the group list again.
- Since the groups with larger weights are placed first in the list, this allows them to get more service than the lower-weight groups at the end of the list.

GRRR – Intergroup scheduling

- Given a group G_i whose weight is x times larger than the group weight of the next group G_{i+1} in the group list, GR3 will select group G_i x times for every time that it selects G_{i+1} in the group list to provide proportional share allocation among groups
- Intergroup Scheduling algorithm maintains the total work done by group G_i in a variable W_i. An index i to tracks the current group and is initialized to 1.
- The scheduling algorithm then executes the following simple routine:

```
1 C ← INTRAGROUP-SCHEDULE($G_i$)
2 $W_i$ ← $W_i$ + 1
3 if i < g and $W_i+1 \,/\, W_{i+1}+1 > \Phi_i/\Phi_{i+1}$ ->(1)
4       then i ← i + 1
5       else i ← 1
6 return C
```

Where,
ϕ_C The weight assigned to client C.
W_G The group work of group G.
g The number of groups.
C_j Client j. (also called 'task' j)
G_i i'th group in the list ordered by weight.

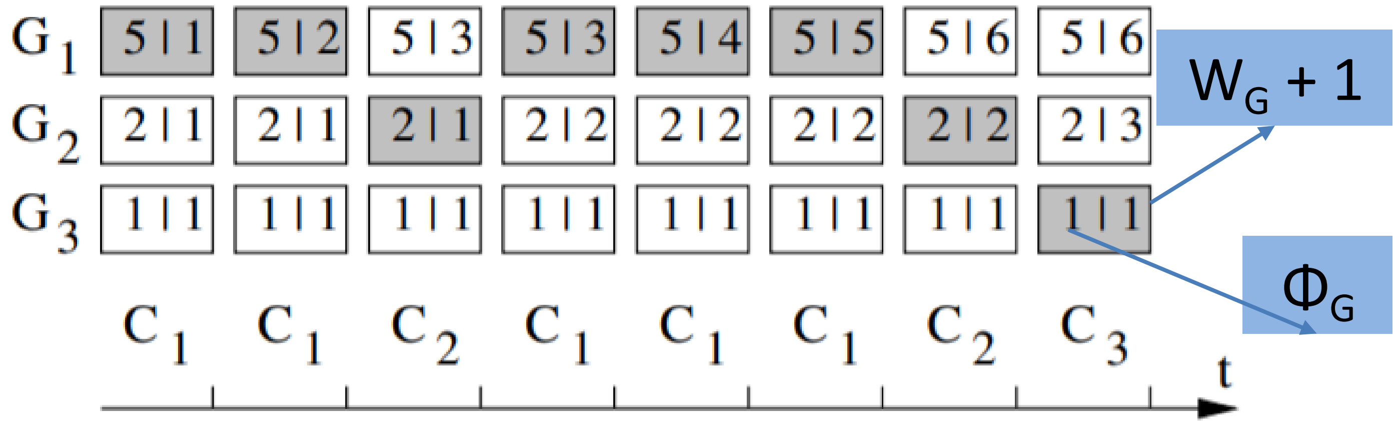
GRRR – Intergroup scheduling

- Let's consider 3 clients C1, C2, and C3, which have weights of 5, 2, and 1, respectively.

- The GRRR grouping strategy would place each Ci in group Gi, ordering the groups by weight: G1, G2, & G3 have orders 2, 1 and 0 and weights of 5, 2, & 1 respectively.

GRRR – Intergroup scheduling

- Let's consider 3 clients C1, C2, and C3, which have weights of 5, 2, and 1, respectively.
- The GRRR grouping strategy would place each Ci in group Gi, ordering the groups by weight: G1, G2, & G3 have orders 2, 1 and 0 and weights of 5, 2, & 1 respectively.

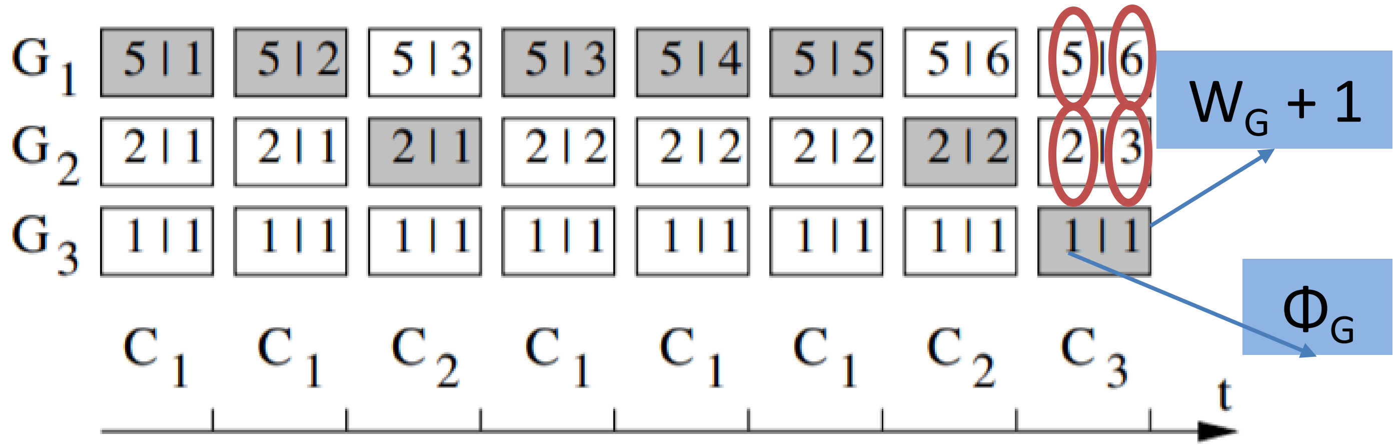


At each time step, the shaded box contains the pair $\Phi_G \mid W_G + 1$ for the group G before it is selected.

GRRR – Intergroup scheduling

$$\frac{W_1+1}{W_2+1} = \frac{6}{3} = 2 < \frac{\Phi_1}{\Phi_2} = \frac{5}{2} = 2.5$$

so GR3 would select G1 again and run client C1.



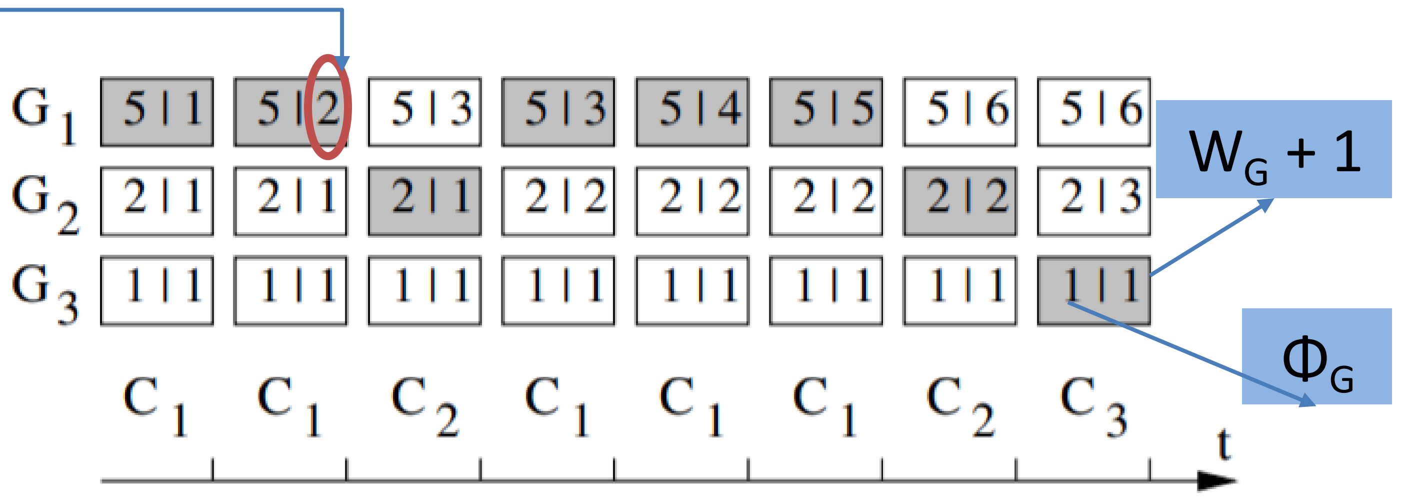
At each time step, the shaded box contains the pair $\Phi_G \mid W_G + 1$ for the group G before it is selected.

GRRR – Intergroup scheduling

$$\frac{W_1+1}{W_2+1} = \frac{6}{3} = 2 < \frac{\Phi_1}{\Phi_2} = \frac{5}{2} = 2.5$$

so GR3 would select G1 again and run client C1.

After running C1, G1's work would be 2 so that the inequality in (1) would hold and GR3 would then move on to the next group G2 and run client C2.



At each time step, the shaded box contains the pair $\Phi_G \mid W_G + 1$ for the group G before it is selected.

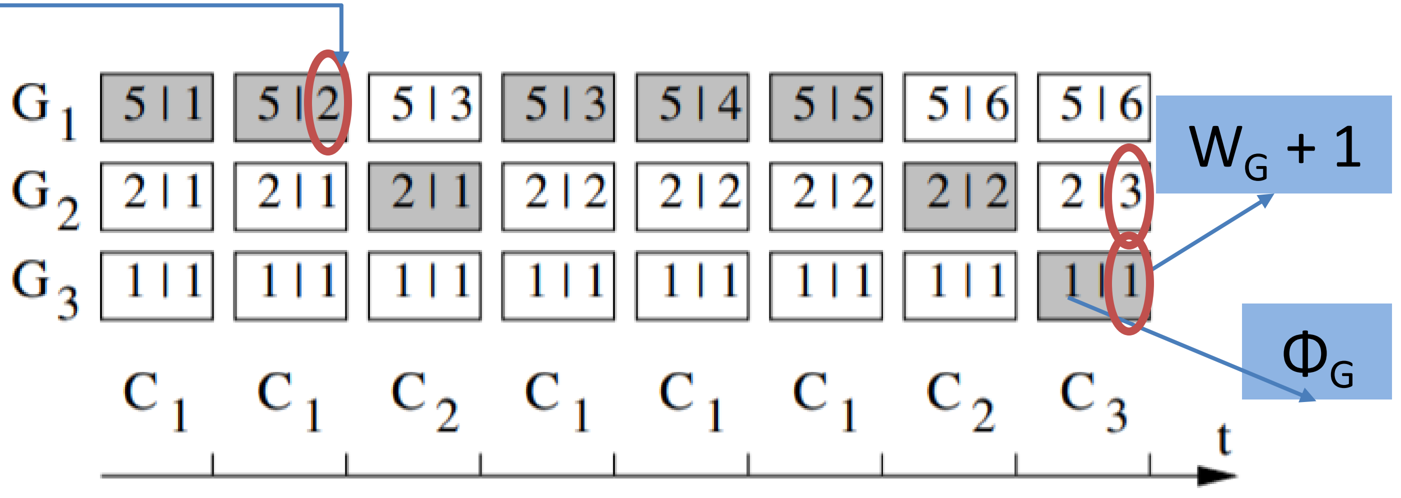
Note:- In this example, each group has only one client so there is no intragroup scheduling.

GRRR – Intergroup scheduling

$\frac{W_1+1}{W_2+1} = \frac{6}{3} = 2 < \frac{\Phi_1}{\Phi_2} = \frac{5}{2} = 2.5$

so GR3 would select G1 again and run client C1.

After running C1, G1's work would be 2 so that the inequality in (1) would hold and GR3 would then move on to the next group G2 and run client C2.



G_1: 5 | 1, 5 | 2, 5 | 3, 5 | 3, 5 | 4, 5 | 5, 5 | 6, 5 | 6
G_2: 2 | 1, 2 | 1, 2 | 1, 2 | 2, 2 | 2, 2 | 2, 2 | 2, 2 | 3
G_3: 1 | 1, 1 | 1, 1 | 1, 1 | 1, 1 | 1, 1 | 1, 1 | 1, 1 | 1
C_1 C_1 C_2 C_1 C_1 C_1 C_2 C_3
t
$W_G + 1$
Φ_G

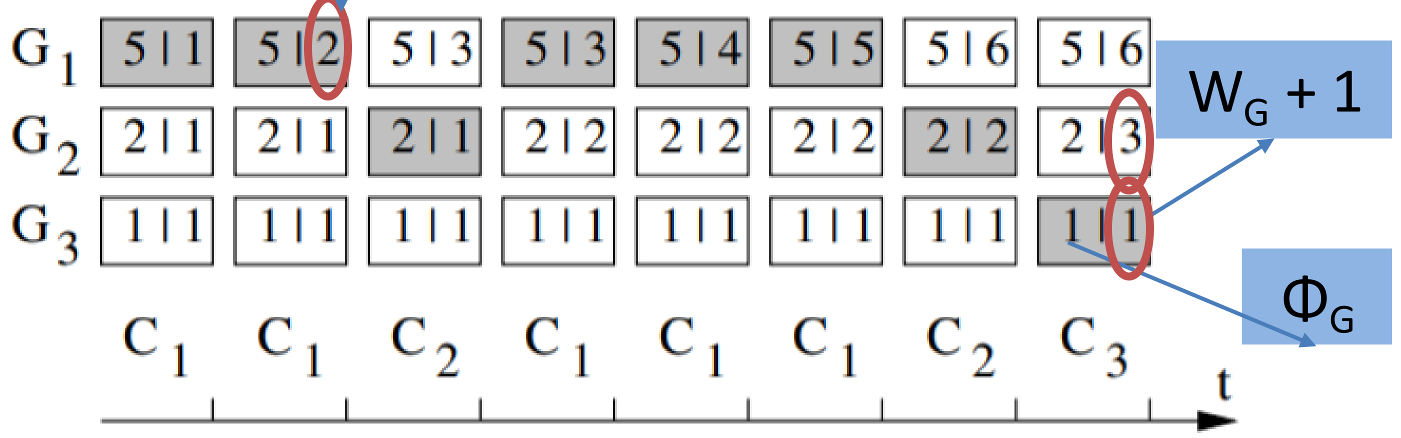
At each time step, the shaded box contains the pair $\Phi_G \mid W_G + 1$ for the group G before it is selected.

Based on (1), $\frac{W_2+1}{W_3+1} = \frac{4}{2} = 2 \leq \frac{\Phi_2}{\Phi_3} = \frac{2}{1} = 2$, so GR3 would reset the current group to the largest weight group G1 and run client C1.

GRRR – Intergroup scheduling

$\frac{W_1+1}{W_2+1} = \frac{6}{3} = 2 < \frac{\Phi_1}{\Phi_2} = \frac{5}{2} = 2.5$ so GR3 would select G1 again and run client C1.

After running C1, G1's work would be 2 so that the inequality in (1) would hold and GR3 would then move on to the next group G2 and run client C2.



At each time step, the shaded box contains the pair Φ_G | $W_G + 1$ for the group G before it is selected.

Based on (1), $\frac{W_2+1}{W_3+1} = \frac{4}{2} = 2 \leq \frac{\Phi_2}{\Phi_3} = \frac{2}{1} = 2$, so GR3 would reset the current group to the largest weight group G1 and run client C1.

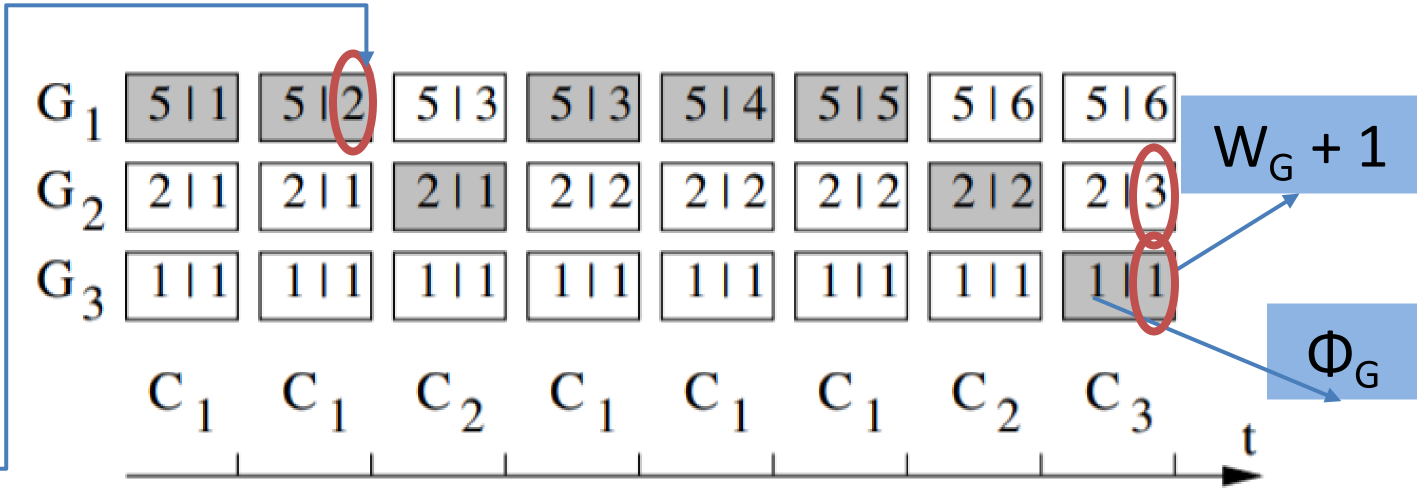
Based on (1), C1 would be run for three time quanta before selecting G2 again to run client C2.

GRRR – Intergroup scheduling

$\frac{W_1+1}{W_2+1} = \frac{6}{3} = 2 < \frac{\Phi_1}{\Phi_2} = \frac{5}{2} = 2.5$

so GR3 would select G1 again and run client C1.

After running C1, G1's work would be 2 so that the inequality in (1) would hold and GR3 would then move on to the next group G2 and run client C2.



G_1: 5 | 1, 5 | 2, 5 | 3, 5 | 3, 5 | 4, 5 | 5, 5 | 6, 5 | 6
G_2: 2 | 1, 2 | 1, 2 | 1, 2 | 2, 2 | 2, 2 | 2, 2 | 2, 2 | 3
G_3: 1 | 1, 1 | 1, 1 | 1, 1 | 1, 1 | 1, 1 | 1, 1 | 1, 1 | 1
C_1 C_1 C_2 C_1 C_1 C_1 C_2 C_3
t
$W_G + 1$
Φ_G

At each time step, the shaded box contains the pair Φ_G | $W_G + 1$ for the group G before it is selected.

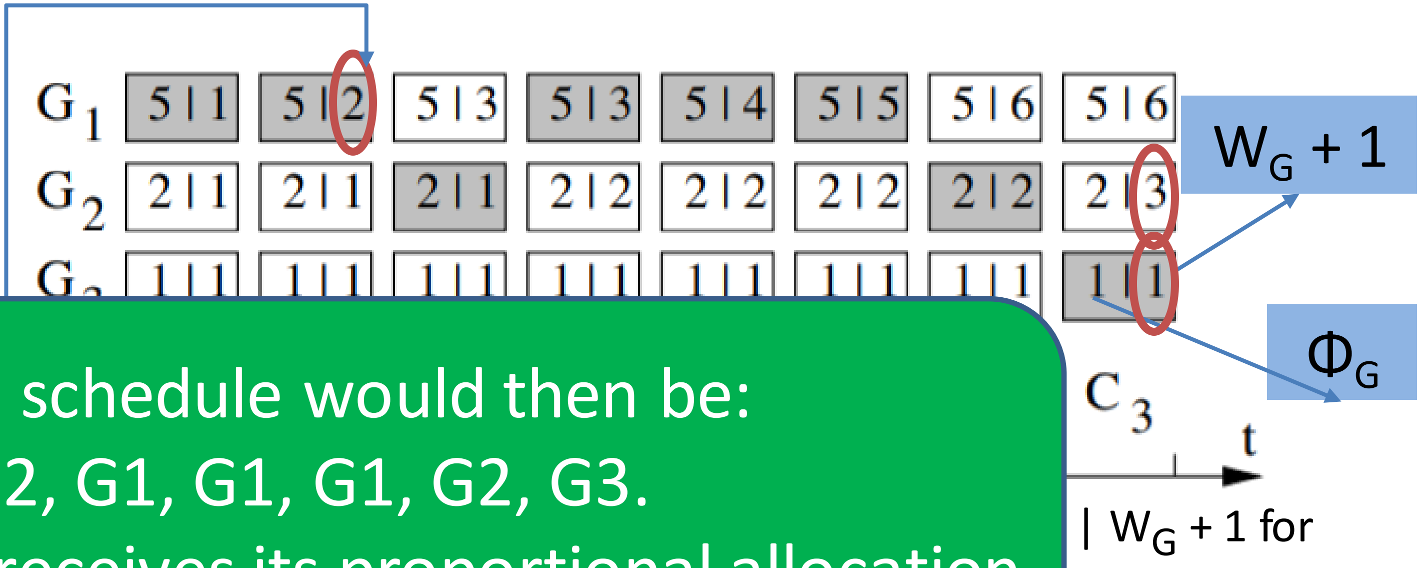
Based on (1), $\frac{W_2+1}{W_3+1} = \frac{4}{2} = 2 \leq \frac{\Phi_2}{\Phi_3} = \frac{2}{1} = 2$, so GR3 would reset the current group to the largest weight group G1 and run client C1.

Based on (1), C1 would be run for three time quanta before selecting G2 again to run client C2.

After running C2 the second time, W2 would increase such that $\frac{W_2+1}{W_3+1} = 3 > \frac{\Phi_2}{\Phi_3} = 2$, so GR3 would then move on to the last group G3 & run client C3.

GRRR – Intergroup scheduling

$\frac{W_1+1}{W_2+1} = \frac{6}{3} = 2 < \frac{\Phi_1}{\Phi_2} = \frac{5}{2} = 2.5$
so GR3 would select G1 again and run client C1.



After running C1 … that the inequal… GR3 would then… group G2 and ru…

| $W_G + 1$ for

The resulting schedule would then be:
G1, G1, G2, G1, G1, G1, G2, G3.
Each group therefore receives its proportional allocation in accordance with its respective group weight.

Based on (1), W… W_3+1 … 2 … Φ_3 … 1 … and run client C1. …eight group G1

Based on (1), C1 would be run for three time quanta before selecting G2 again to run client C2.

After running C2 the second time, W2 would increase such that $\frac{W_2+1}{W_3+1} = 3 > \frac{\Phi_2}{\Phi_3} = 2$, so GR3 would then move on to the last group G3 & run client C3.

GRRR – Intragroup scheduling

- From the selected group, a client is selected to run in a round-robin manner that accounts for its weight and previous execution history.
- The GR3 intragroup scheduling algorithm selects a client from the selected group.
- All clients within a group have weights within a factor of two, and all client weights in a group G are normalized with respect to the minimum possible weight, ϕ_{min}^{G} = $2^{\sigma G}$ for any client in the group.
- GR3 then effectively traverses through a group's queue in round-robin order, allocating each client its normalized weight worth of time quanta.
- GR3 keeps track of subunitary fractional time quanta that cannot be used and accumulates them in a deficit value for each client.
- Hence, each client is assigned either one or two time quanta, based on
- the client's normalized weight and its previous allocation.

GRRR – Intragroup scheduling

- More specifically, the GR3 intragroup scheduler considers the scheduling of clients in rounds.
- A round is one pass through a group G’s run queue of clients from beginning to end.
- The group run queue does not need to be sorted in any manner.
- During each round, the GR3 intragroup algorithm considers the clients in round-robin order and executes the following simple routine:

```
$C \leftarrow G[k]$, **k is the current position in the round**
if $D_C < 1$
 then $k \leftarrow (k + 1) \bmod |G|$
      $C \leftarrow G[k]$
      $D_C \leftarrow D_C + \phi_C \mid \phi_{min}^{G}$
 $D_C \leftarrow D_C - 1$
 Return C
```

Where,
ϕ_C The weight assigned to client C.
D_C The deficit of C.
G(C) The group to which C belongs.
σG The order of group G.
ϕ_{min}^{G} Lower bound for client weights in G: $2^{\sigma G}$.
C_j Client j. (also called ’task’ j)
|G| The number of clients in group G.

GRRR – Intragroup scheduling

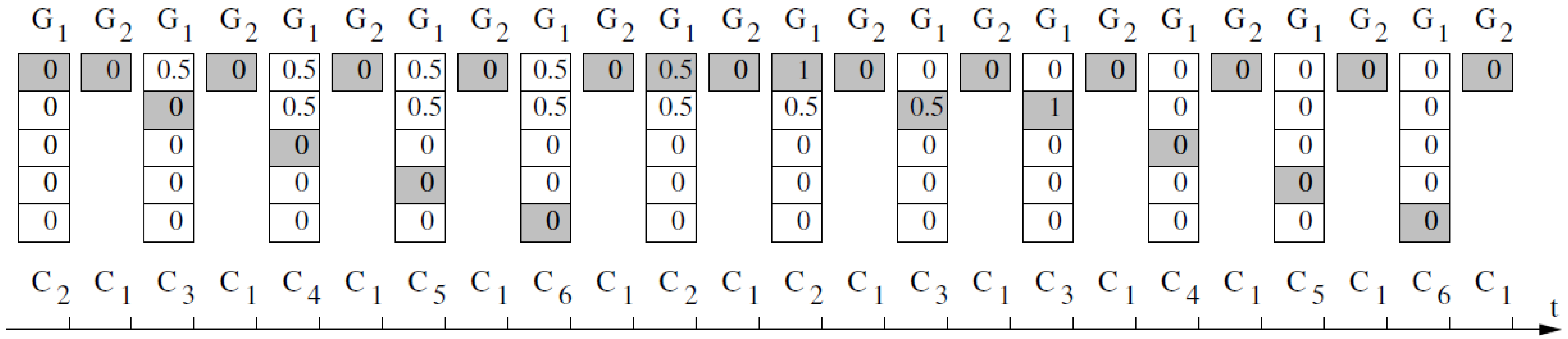
- For each runnable client C, the scheduler determines the maximum no. of time quanta that the client can be selected to run in this round as :-

$$\left\lfloor \frac{\varphi_C}{\varphi_{min}^G} + D_C(r-1) \right\rfloor . D_C(r)$$

- The deficit of client C after round r, is the time quantum fraction left over after round r: $D_C(r)$ = $\frac{\varphi_C}{\varphi_{min}^G} + D_C(r-1)$ - $\left\lfloor \frac{\varphi_C}{\varphi_{min}^G} + D_C(r-1) \right\rfloor$ with $D_C(0)$ = $\frac{\varphi_C}{\varphi_{min}^G}$
- Thus, in each round, C is allotted one time quantum plus any additional leftover from the previous round, and $D_C(r)$ keeps track of the amount of service that C missed because of rounding down its allocation to whole time quanta.
- if a client is allotted two time quanta, it first executes for one time quantum & then executes for the second time quantum the next time the intergroup scheduler selects its respective group again

GRRR – Intragroup scheduling

- Lets consider and an example with six clients C1 through C6 with weights 12, 3, 3, 2, 2, and 2, respectively.
- The six clients will be put in two groups G1 and G2 with respective group order 1 and 3 as follows: G1 = {C2, C3, C4, C5, C6} and G2 = {C1}.
- The weight of the groups are $\phi_1 = \phi_2 = 12$
- GR3 intergroup scheduling will consider the groups in this order: G1, G2, G1, G2, G1, G2, G1, G2, G1, G2, G1, G2.



At each time step, the shaded box contains the deficit of the client before it is run.

GRRR – Intragroup scheduling

- G2 will schedule client C1 every time G2 is considered for service since it has only 1 client
- Since ϕ_{min}^{G1} = 2, the normalized weights of clients C2, C3, C4, C5, and C6 are 1.5, 1.5, 1, 1
- In the b a result, the intra um during round 1
- After th 0, 0, and 0.
- In the beginning of round 2, each client gets another $\frac{\varphi_i}{\varphi_{min}^{G1}}$ allocation, plus any deficit from the first round.
- As a result, the intragroup scheduler will select clients C2, C3, C4, C5, and C6 to run in order for 2, 2, 1, 1, and 1 time quanta, respectively, during round 2.

The resulting schedule would then be:
C2, C1, C3, C1, C4, C1, C5, C1, C6, C1, C2, C1, C2, C1, C3, C1, C3, C1, C4, C1, C5, C1, C6, C1.

GRRR Multiprocessor Extensions (GR^{3}MP)

- GR^{3}MP uses the same GR^3 data structure, an ordered list of groups, each containing clients whose weights are within a factor of two from each other.

- When a processor needs to be scheduled,GR^{3}MP selects the client that would run next under GR^3, essentially scheduling multiple processors from its central run queue as GR^3 schedules a single processor.

GRRR Multiprocessor Extensions (GR^{3}MP)

- Issue with this Approach:
    - By applying a uniprocessor algorithm on a multiprocessor system, each client can only run on one processor at any given time.
    - As a result, GR3MP cannot select a client to run that is already running on another processor even if GR^3 would schedule that client in the uniprocessor case.
    - For example, if GR^3 would schedule the same client consecutively, GR^{3}MP cannot schedule that client consecutively on another processor if it is still running.

GRRR Multiprocessor Extensions (GR^{3}MP)

- Solution:
  - To handle this situation while maintaining fairness, GR3MP introduces the notion of a **frontlog**.
  - The frontlog F_C for some client C running on a processor $\mathscr{p}^k$ (C = C($\mathscr{p}^k$)) is defined as the number of time quanta for C accumulated as C gets selected by GR3 and cannot run because it is already running on $\mathscr{p}^k$.
  - The frontlog F_C is then queued up on $\mathscr{p}^k$.

GRRR Multiprocessor Extensions (GR^{3}MP)

```
MP-SCHEDULE($p^k$)
1  C ← C($p^k$)                              ->Client just run
2  if C = NIL
3      then if N <P
4          then return NIL                   ->Idle
5      else if $F_c$ > 0
6          then $F_c$ ← $F_c$ − 1
7              return C
8  C ← INTERGROUP-SCHEDULE()
9  while $p^k$ s.t. C = C($p$)
10         do $F_c$ ← $F_c$ + 1
11         C ← INTERGROUP-SCHEDULE()
12 return C
```

Whenever a processor finishes running a client for a time quantum, GR$_3$MP checks whether the client has a non-zero frontlog,

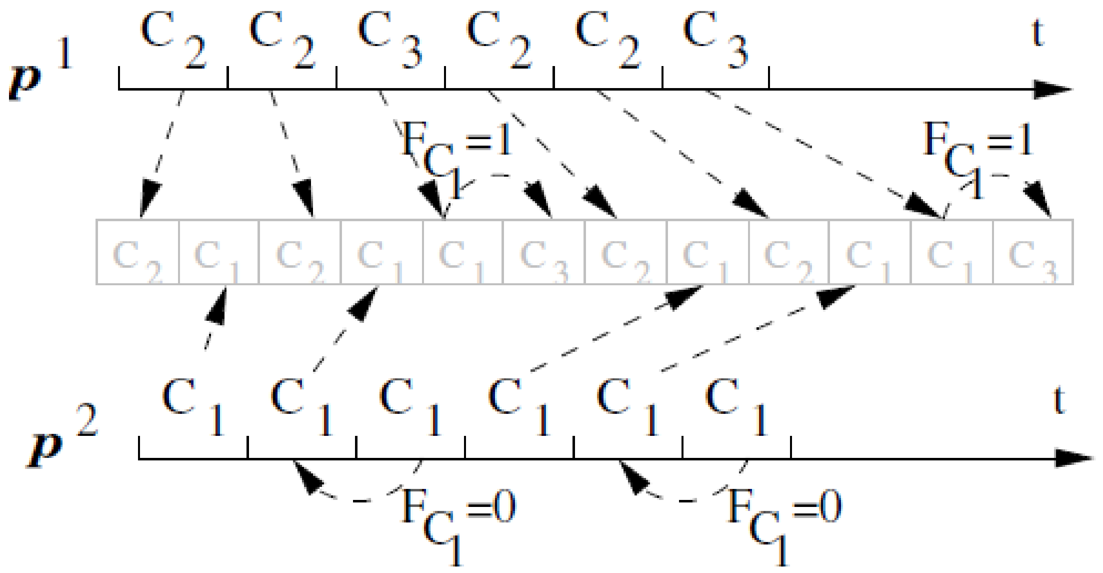
if $F_c > 0$, continues running the client for another time quantum and decrements its frontlog by one, without consulting the central queue.

If the selected client is already running on some other processor, increase its frontlog and repeat the GR3 scheduling.

each time incrementing the frontlog of the selected client, until we find a client that is not currently running.

GRRR Multiprocessor Extensions (GR^{3}MP)

- Lets consider an example - on a dual-processor system with three clients C1, C2, and C3 of weights 3, 2, and 1, respectively.
- C1 and C2 will then be part of the order 1 group (assume C2 is before C1 in the round-robin queue of this group), whereas C3 is part of the order 0 group. The GR3 schedule is C2,C1, C2, C1, C1, C3.



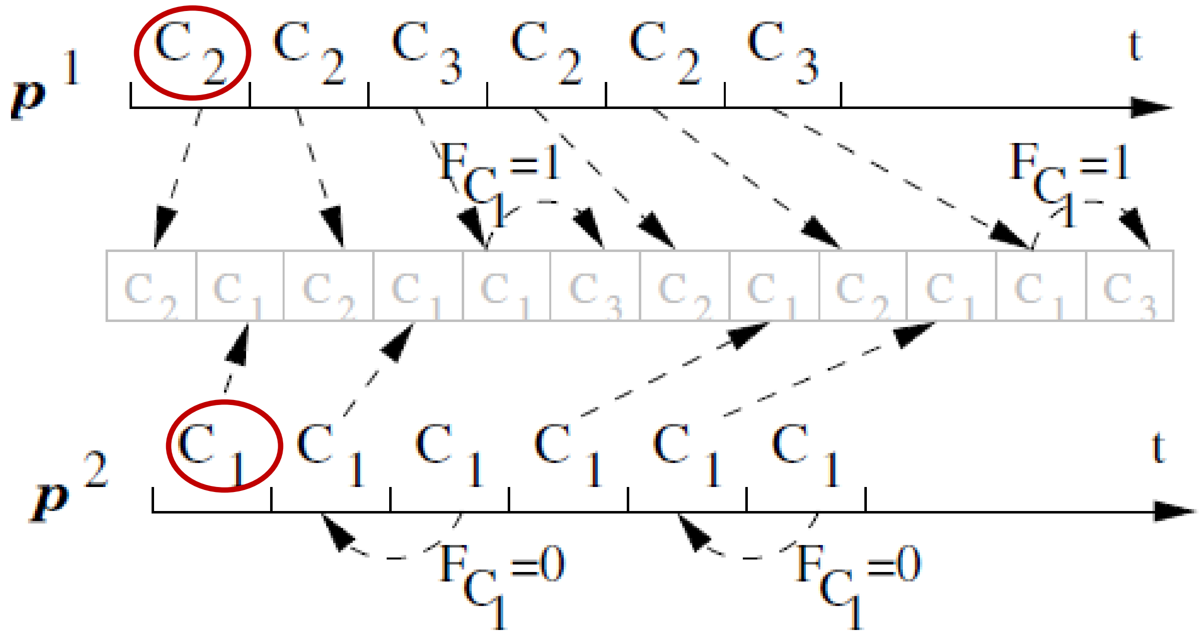
p^1
C_2 C_2 C_3 C_2 C_2 C_3
t
$F_{C_1}=1$
$F_{C_1}=1$
C_2 C_1 C_2 C_1 C_1 C_3 C_2 C_1 C_2 C_1 C_1 C_3
p^2
C_1 C_1 C_1 C_1 C_1 C_1
t
$F_{C_1}=0$
$F_{C_1}=0$

P	Number of processors.
$\mathscr{p}^k$	Processor k.
C($\mathscr{p}$)	Client running on processor $\mathscr{p}$.
F_c	Frontlog for client C.

The two processors schedule either from the central queue, or use the frontlog mechanism when the task is already running.

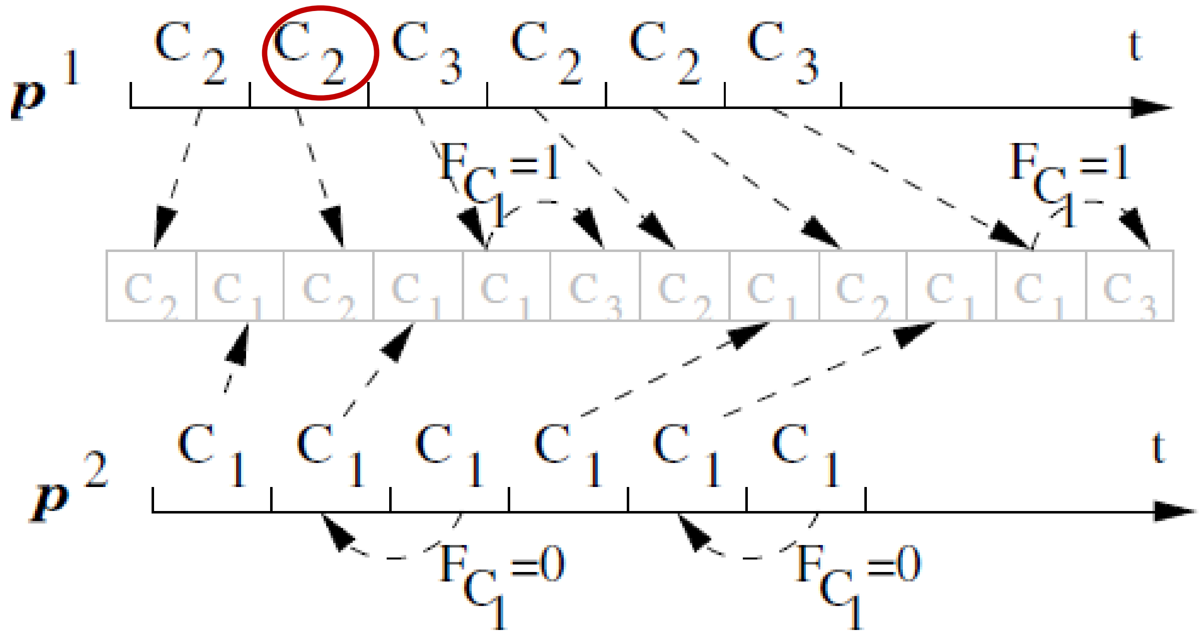
GRRR Multiprocessor Extensions (GR^{3}MP)

- $\wp^1$ will select C2 to run, and $\wp^2$ selects C1.



GRRR Multiprocessor Extensions (GR^{3}MP)

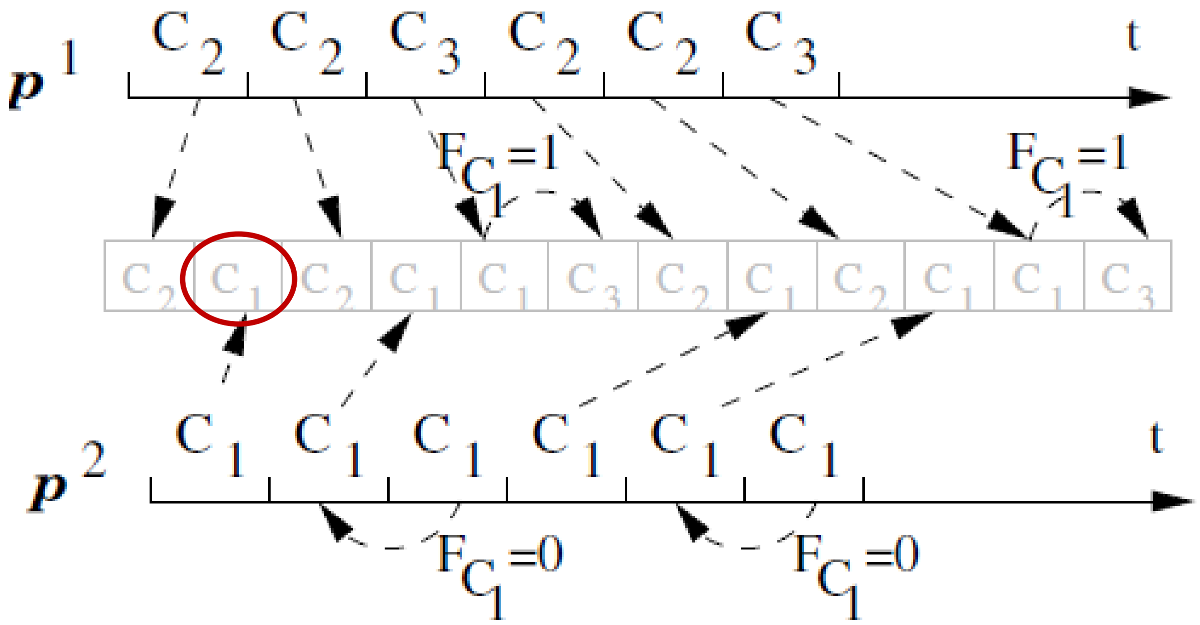
- $\wp^1$ will select C2 to run, and $\wp^2$ selects C1.
- When $\wp^1$ finishes, according to GR3, it will select C2



p^1
C_2 C_2 C_3 C_2 C_2 C_3
t
$F_{C_1}=1$
$F_{C_1}=1$
C_2 C_1 C_2 C_1 C_1 C_3 C_2 C_1 C_2 C_1 C_1 C_3
p^2
C_1 C_1 C_1 C_1 C_1 C_1
t
$F_{C_1}=0$
$F_{C_1}=0$

GRRR Multiprocessor Extensions (GR^{3}MP)

- p^1 will select C2 to run, and p^2 selects C1.
- When p^1 finishes, according to GR3, it will select C2
- When p^1 again selects the next GR3 client, which is C1, it finds that it is already running on p^2 and thus we set F_{C1} = 1 and select the next client, which is C3, to run on p^1.



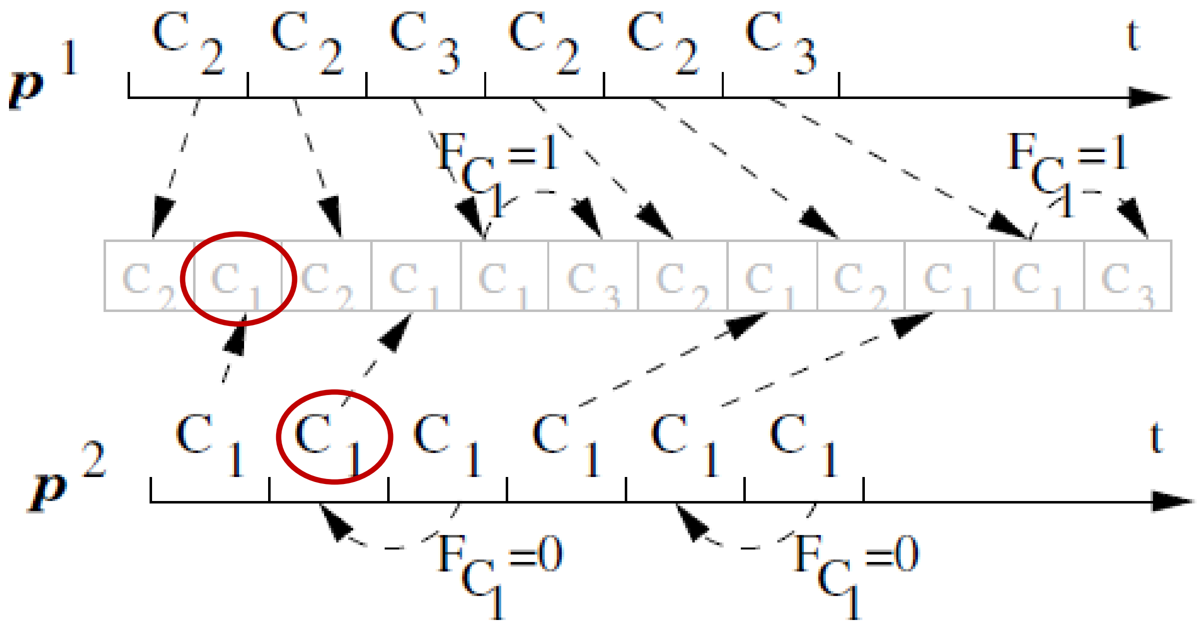
p^1
C_2 C_2 C_3 C_2 C_2 C_3
t
$F_{C_1}=1$
$F_{C_1}=1$
C_2 C_1 C_2 C_1 C_1 C_3 C_2 C_1 C_2 C_1 C_1 C_3
p^2
C_1 C_1 C_1 C_1 C_1 C_1
t
$F_{C_1}=0$
$F_{C_1}=0$

GRRR Multiprocessor Extensions (GR^{3}MP)

- p^1 will select C2 to run, and p^2 selects C1.

- When p^1 finishes, according to GR3, it will select C2

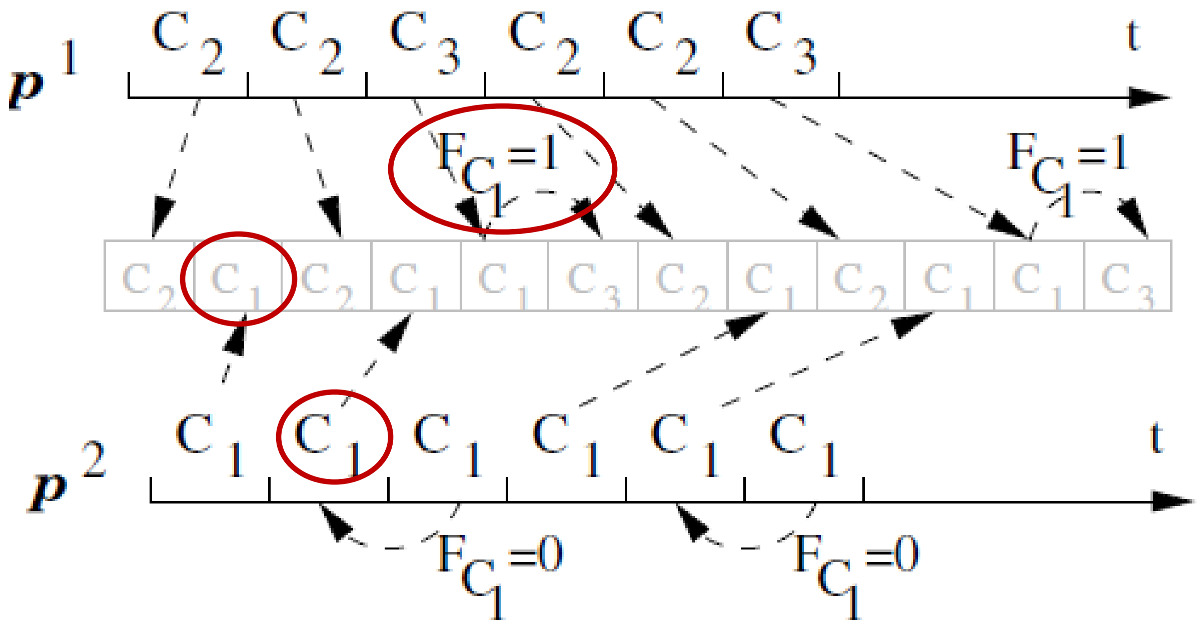
- When p^1 again selects the next GR3 client, which is C1, it finds that it is already running on p^2 and thus we set F_{C1} = 1 and select the next client, which is C3, to run on p^1.



p^1
C_2 C_2 C_3 C_2 C_2 C_3
t
$F_{C_1}=1$
$F_{C_1}=1$
C_2 C_1 C_2 C_1 C_1 C_3 C_2 C_1 C_2 C_1 C_1 C_3
p^2
C_1 C_1 C_1 C_1 C_1 C_1
t
$F_{C_1}=0$
$F_{C_1}=0$

GRRR Multiprocessor Extensions (GR^{3}MP)

- p^1 will select C2 to run, and p^2 selects C1.
- When p^1 finishes, according to GR3, it will select C2
- When p^1 again selects the next GR3 client, which is C1, it finds that it is already running on p^2 and thus we set F_{C1} = 1 and select the next client, which is C3, to run on p^1.

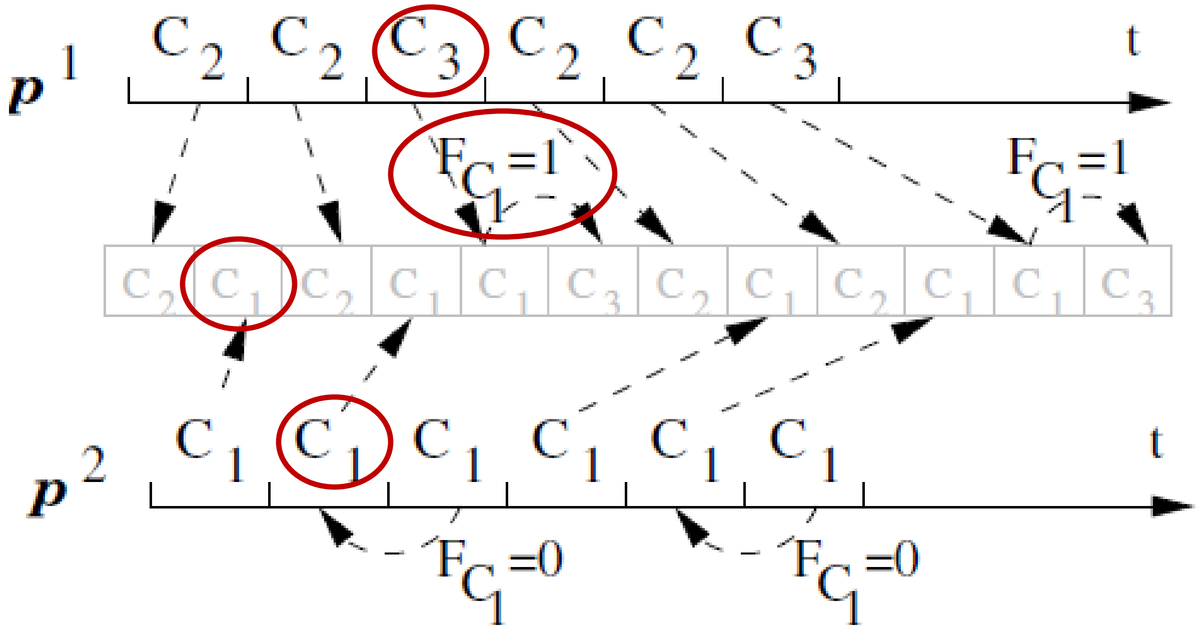

p^1
C_2 C_2 C_3 C_2 C_2 C_3
t
$F_{C_1}=1$
$F_{C_1}=1$
C_2 C_1 C_2 C_1 C_1 C_3 C_2 C_1 C_2 C_1 C_1 C_3
p^2
C_1 C_1 C_1 C_1 C_1 C_1
t
$F_{C_1}=0$
$F_{C_1}=0$

GRRR Multiprocessor Extensions (GR^{3}MP)

- $\wp^1$ will select C2 to run, and $\wp^2$ selects C1.

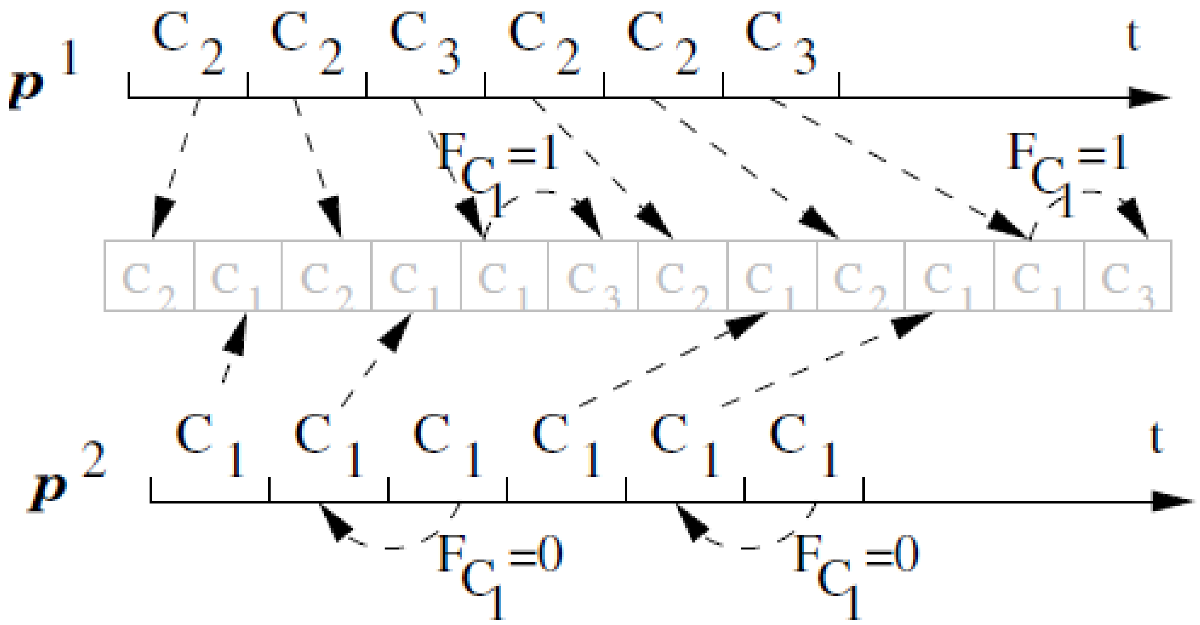
- When $\wp^1$ finishes, according to GR3, it will select C2

- When $\wp^1$ again selects the next GR3 client, which is C1, it finds that it is already running on $\wp^2$ and thus we set $F_{C1} = 1$ and select the next client, which is C3, to run on $\wp^1$.



GRRR Multiprocessor Extensions (GR^{3}MP)

- p^1 will select C2 to run, and p^2 selects C1.
- When p^1 finishes, according to GR3, it will select C2
- When p^1 again selects the next GR3 client, which is C1, it finds that it is already running on p^2 and thus we set F_{C1} = 1 and select the next client, which is C3, to run on p^1.
- When p^2 finishes running C1 for its second time quantum, it finds F_{C1} = 1, sets F_{C1} = 0 and continues running C1 without any scheduling decision on the GR3 queue.



p^1
C_2 C_2 C_3 C_2 C_2 C_3
t
$F_{C_1}=1$
$F_{C_1}=1$
C_2 C_1 C_2 C_1 C_1 C_3 C_2 C_1 C_2 C_1 C_1 C_3
p^2
C_1 C_1 C_1 C_1 C_1 C_1
t
$F_{C_1}=0$
$F_{C_1}=0$

GRRR – Client grouping strategy

- Pros
  - Using this client grouping strategy, GR3 separates scheduling in a way that reduces the need to schedule entities with skewed weight distributions.
  - The client grouping strategy limits the number of groups that need to be scheduled since the number of groups grows at worst logarithmically with the largest client weight.
  - Even a very large 32-bit client weight would limit the number of groups to no more than 32.
  - The client grouping strategy also ensures that all clients within a group have weight within a factor of two.
- Therefore, the intragroup scheduler never needs to schedule clients with skewed weight distributions.
- GR3 groups are simple lists that do not need to be balanced; they do not require any use of more complex balanced tree structures.

TIMER INTERRUPTS

- Timer interrupts is the “engine” which drives scheduling, integrated with other scheduling primitive and constructs.

Interrupt-Driven Scheduling

- Like other contemporary OS, Mach OS also uses the hardware interrupts to usurp control of the CPU from the existing thread.
- Interrupts forces the CPU to “drop everything” on interrupt and longjmp to the interrupt handler (also known as the interrupt service routine, or ISR).
- Apart from asynchronous interrupt, XNU also provides an interrupt that can be triggered in a given time frame for a predictable interrupt source, namely, real time clock, or rtclock.
- This clock is hardware dependent — the Intel architecture uses the local CPU’s APIC for this purpose — and can be configured by the kernel to generate an interrupt after a given number of cycles.
- The interrupt source is often referred to as the ***Timer Interrupt***

Interrupt-Driven Scheduling

- Older versions of XNU triggered the Timer Interrupt a fixed number of times per second, a value referred to as hz.

- This value is globally defined in the BSD portion of the kernel, in bsd/kern/clock.c (now deprecated)

Indian Institute of Technology Kharagpur

Issue with fixed interval in Timer Interrupt

- A timer interrupting the kernel at a fixed interval will cause predictable, but extraneous interrupts.
- Too high a value of hz implies too many unnecessary interrupts.
- On the other hand, too low a value would mean the system is less responsive, as sub-hz delays would only be achievable by a tight loop.
- The old `hertz_tick()` function used in previous versions of OS X is still present, but unused and conditionally compiled only if XNU is compiled with profiling.

Indian Institute of Technology Kharagpur

Solution - A *tick-less* kernel

- A tick-less kernel (similar to Linux (versions 2.6.21 and above), in which on every Timer Interrupt the timer is *reset* to schedule the next interrupt only when the scheduler seems it necessary.

- This means that, on every Timer Interrupt, the interrupt handler has to make a (very quick) pass over the list of pending deadlines, which are primarily sleep timeouts set by threads, act on them, if necessary, and schedule the next Timer Interrupt accordingly.

- More processing in each Timer Interrupt is well worth the savings in spurious interrupts, and the processing can be kept to a minimum by keeping track of only the most exigent deadline.

Indian Institute of Technology Kharagpur

Additional Resources

- "Programming Semantics for Multiprogrammed Computations," by Jack B. Dennis and Earl C. Van Horn (The ACM Conference on Programming Languages and Pragmatics, San Dimas, California, August 1965).
- UNIX Programmers Manual, by K. Thompson and D. M. Ritchie (Bell Laboratories, 1971).

Indian Institute of Technology Kharagpur

Happy Learning