SCS 1312: Operating Systems

Dr. Dinuni Fernando, PhD

Based on Operating System Concepts by A.Silberschatz, P.Galvin, and G.Gagne

CPU Scheduling II

Scheduling algorithms : 5. Multilevel Queue

- Divides the ready queue into multiple priority queues and assigns a different priority to each queue.
- Each queue may have its own scheduling algorithm, processes are assigned to the queue based on their priority.
- Process with the highest priority is scheduled first, if there are multiple processes in the same priority queue, a separate scheduling algorithm within that queue is used.
- Eg: with 3 priority levels
  - Queue 1 (Highest Priority): Time-Critical Processes
  - Queue 2 (Medium Priority): Interactive Processes
  - Queue 3 (Lowest Priority): Batch Processes

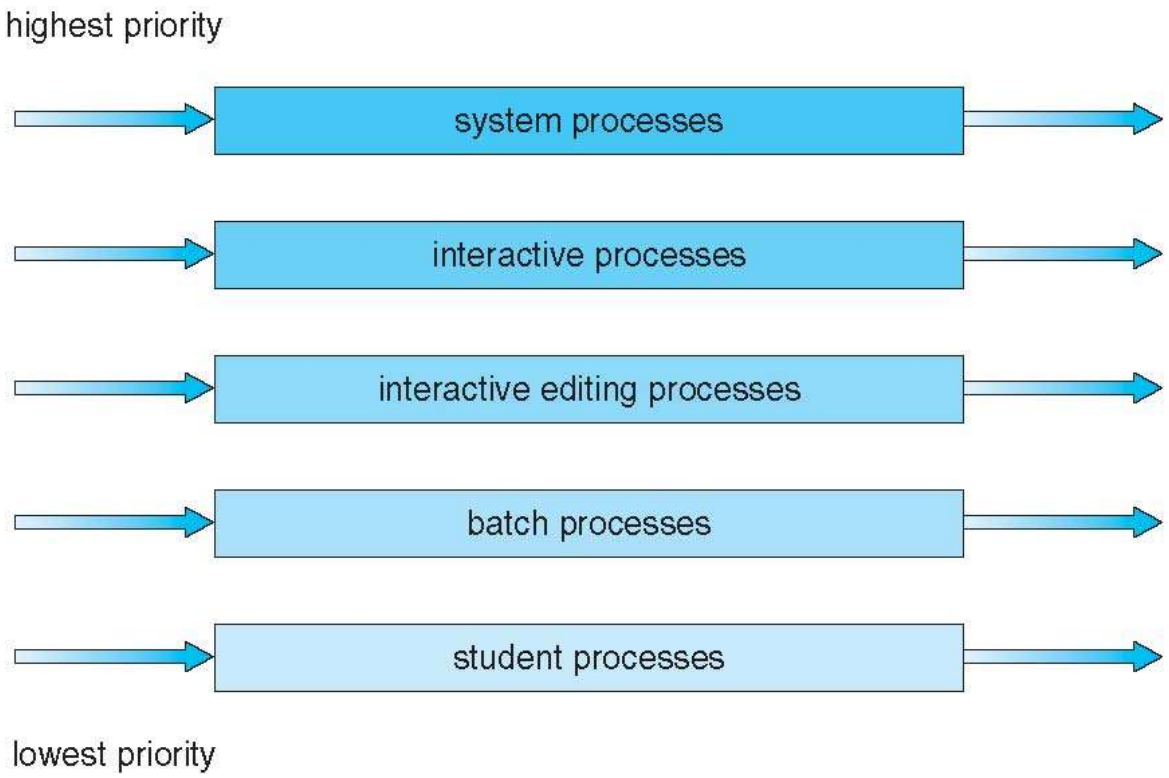
Scheduling algorithms : 5. Multilevel Queue

- Priority assignment
  - Different processes are assigned to different priority levels based on their characteristics and requirements.
  - Time-critical processes may be assigned to the highest priority queue to ensure quick execution.
  - Interactive processes, which require responsiveness, may be placed in a medium-priority queue.
  - Batch processes, which can run in the background, may be assigned to the lowest priority queue.
- Scheduling Algorithms
  - Each queue may use a different scheduling algorithm.
    - Eg: the highest priority queue may use preemptive scheduling, while the lower priority queues may use non-preemptive scheduling.

Scheduling algorithms : 5. Multilevel Queue

- Ready queue is partitioned into separate queues, eg:
  - **foreground** (interactive)
  - **background** (batch)
- Process permanently in a given queue
- Each queue has its own scheduling algorithm:
  - foreground – RR
  - background – FCFS
- Scheduling must be done between the queues:
  - Fixed priority scheduling; (i.e., serve all from foreground then from background). Possibility of starvation.
  - Time slice – each queue gets a certain amount of CPU time which it can schedule amongst its processes; i.e., 80% to foreground in RR
  - 20% to background in FCFS

Multilevel queue scheduling



Scheduling algorithms : 6. Multilevel Feedback Queue

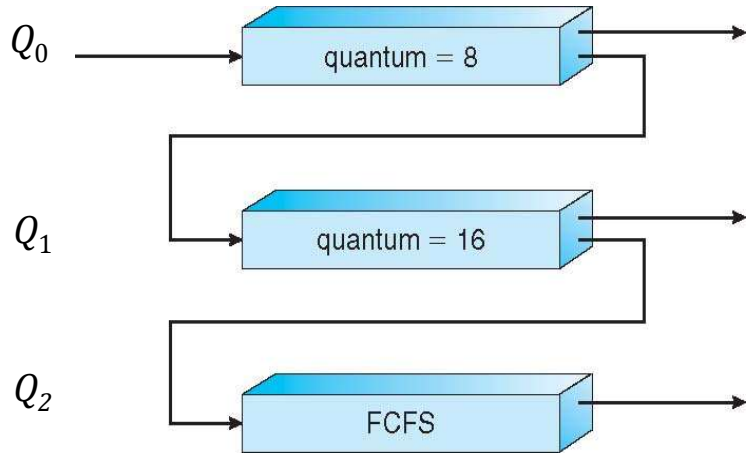
- An extension of the multilevel queue scheduling algorithm.
- It incorporates the concept of dynamically changing priorities based on the behavior of processes.
- Processes can move between different priority queues based on their recent CPU usage patterns.
- This allows the scheduler to adapt to the varying needs of different types of processes over time.
- Priority assignment :
  - All processes are placed in the highest-priority queue.
  - If a process completes its time quantum without blocking or termination, it is moved to the next lower-priority queue.
  - If a process uses its entire time quantum in the lowest-priority queue, it is moved back to the highest-priority queue.

Scheduling algorithms : 6. Multilevel Feedback Queue

- Scheduling algorithm:
  - Each queue may use a different scheduling algorithm (e.g., Round Robin) with a specific time quantum assigned to it.
  - Processes that exhibit CPU-bound behavior are likely to move down the priority levels, while processes with I/O-bound behavior may move up.
- Multilevel-feedback-queue scheduler defined by the following parameters:
  - number of queues
  - scheduling algorithms for each queue
  - method used to determine when to upgrade a process
  - method used to determine when to demote a process
  - method used to determine which queue a process will enter when that process needs service

Example of Multilevel Feedback Queue

- Three queues:
  - Q_0 – RR with time quantum 8 milliseconds
  - Q_1 – RR time quantum 16 milliseconds
  - Q_2 – FCFS



- Scheduling
  - A new job enters queue Q_0
    - When it gains CPU, job receives 8 milliseconds
    - If it does not finish in 8 milliseconds, job is moved to queue Q_1
  - At Q_1 receives 16 additional milliseconds
    - If it still does not complete, it is preempted and moved to queue Q_2

Scheduling algorithms : 7. Thread Scheduling

- Modern operating systems use kernel level threads and not processes.
  - OS schedules them on CPU.
- User-level threads are managed by a thread library and kernel is unaware of them.
  - To run them on CPU, user-level threads must be mapped to kernel level thread.
- To schedule kernel-level threads on CPU, kernel uses system content scope (SCS).
  - Competition for the CPU with SCS scheduling takes place among all threads in the system using one-to-one model.
- Process contention scope (PCS) done according to priority. Scheduler selects highest priority runnable thread .
  - PCS preempt currently running thread in favor of a higher priority thread.

Pthread Scheduling

- Scheduling treads in a multithreaded program using the POSIX threads library (pthread).
- POSIX thread standard defines a set of APIs for creating, synchronizing and managing threads in a Unix-like operating system.
- Thread scheduling – involves determining the order in which threads are executed on a CPU.
  - OS's scheduler is responsible for making these decisions based on scheduling policies and algorithm.

Pthread Scheduling

- In a pthreads-based program, multiple threads can be created within a single process using the pthread_create function.
- API allows specifying either PCS or SCS during thread creation
  - PTHREAD_SCOPE_PROCESS schedules threads using PCS scheduling
  - PTHREAD_SCOPE_SYSTEM schedules threads using SCS scheduling
- Can be limited by OS – Linux and Mac OS X only allow PTHREAD_SCOPE_SYSTEM

Pthread scheduling API

```
include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5
int main(int argc, char *argv[]) {
   int i, scope;
   pthread_t tid[NUM THREADS];
   pthread_attr_t attr;
   /* get the default attributes */
   pthread_attr_init(&attr);
   /* first inquire on the current scope */
   if (pthread_attr_getscope(&attr, &scope) != 0)
      fprintf(stderr, "Unable to get scheduling scope\n");
   else {
      if (scope == PTHREAD_SCOPE_PROCESS)
         printf("PTHREAD_SCOPE_PROCESS");
      else if (scope == PTHREAD_SCOPE_SYSTEM)
         printf("PTHREAD_SCOPE_SYSTEM");
      else
         fprintf(stderr, "Illegal scope value.\n");
   }
```

Pthread scheduling API

```
/* set the scheduling algorithm to PCS or SCS */
   pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
   /* create the threads */
   for (i = 0; i < NUM_THREADS; i++)
      pthread_create(&tid[i],&attr,runner,NULL);
   /* now join on each thread */
   for (i = 0; i < NUM_THREADS; i++)
      pthread_join(tid[i], NULL);
}
/* Each thread will begin control in this function */
void *runner(void *param)
{}
   /* do some work ... */
   pthread_exit(0);
```

Scheduling algorithms : 8. Multi-Processor Scheduling

- For available multiple CPUs – load sharing : multiple threads may run in parallel.
- Multiprocessor – system provide multiple processors where each processor contained one single-core CPU.
- Multiprocessor examples
  - Multicore CPUs
  - Multithreaded cores
  - NUMA system
  - Heterogeneous multiprocessing

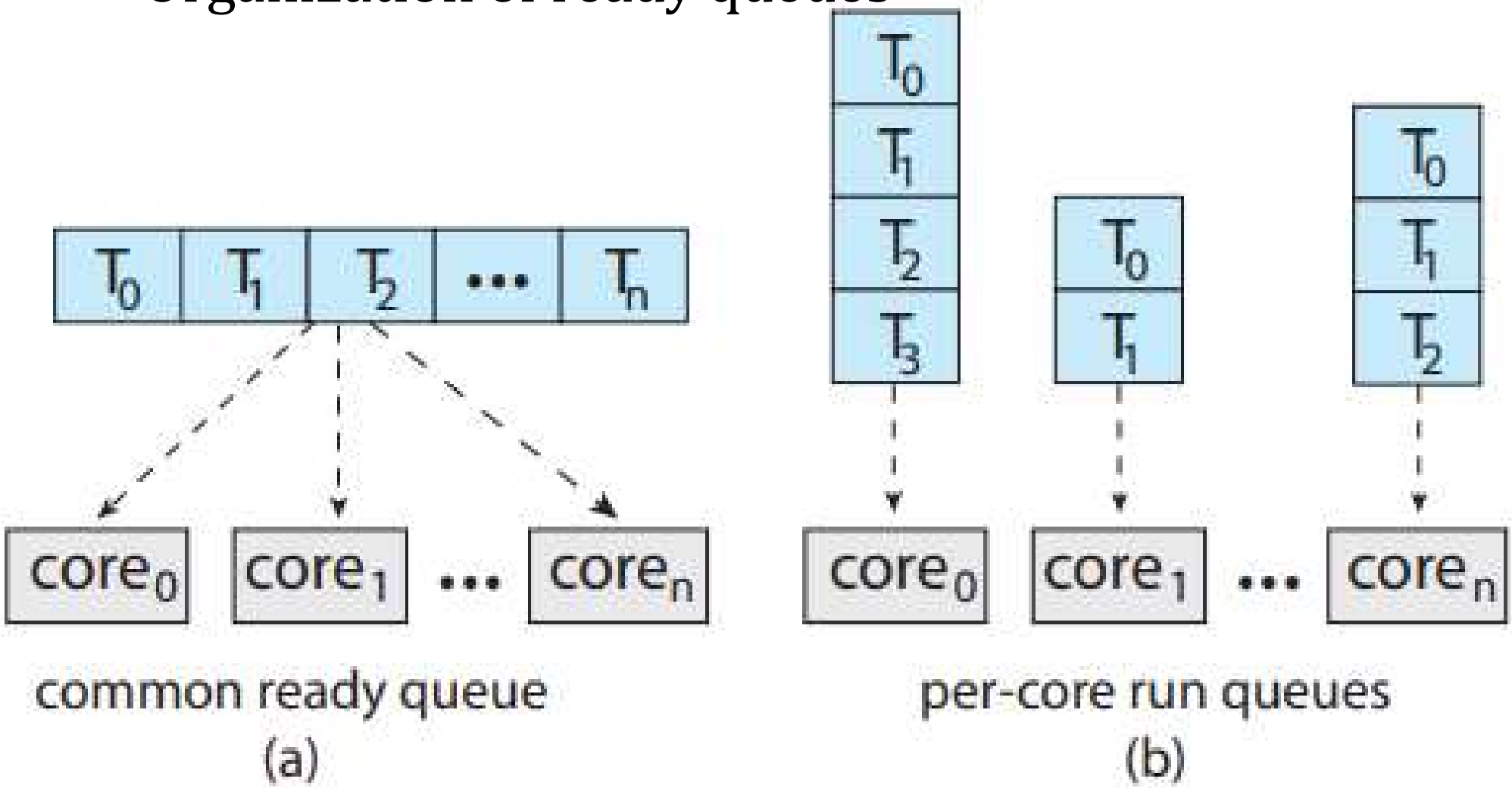
Feature	Multiprocessor	Multicore CPU	Multithreaded Core	NUMA System	Heterogeneous Multiprocessing
Processing Units	Multiple CPUs	Multiple Cores	Single Core, Multiple Threads	Multiple CPUs with Memory Segments	Mix of Performance & Efficiency Cores
Parallelism	High	Medium to High	Low to Medium	High	Adaptive
Memory Model	Shared or Distributed	Shared	Shared	Non-Uniform	Shared or Dedicated
Use Case	Servers, HPC	Consumer & Server CPUs	Workstation, Cloud Computing	Large-scale computing	Mobile, AI, Power-Efficient Computing

Scheduling algorithms : 8. Multi-Processor Scheduling

Approaches for multiple-processor scheduling

- All system activates including scheduling, I/O processing is handled by a single processor – a master server. Other processors execute only user code.
- **Asymmetric multiprocessing** – only one processor accesses the system data structures, alleviating the need for data sharing
- **Symmetric multiprocessing (SMP)** – each processor is self-scheduling, all processes in common ready queue, or each has its own private queue of ready processes.
  - Currently, most common
  - Two possible strategies for organizing threads
    - All threads may be in a common ready queue
    - Each processor may have its own private queue of threads

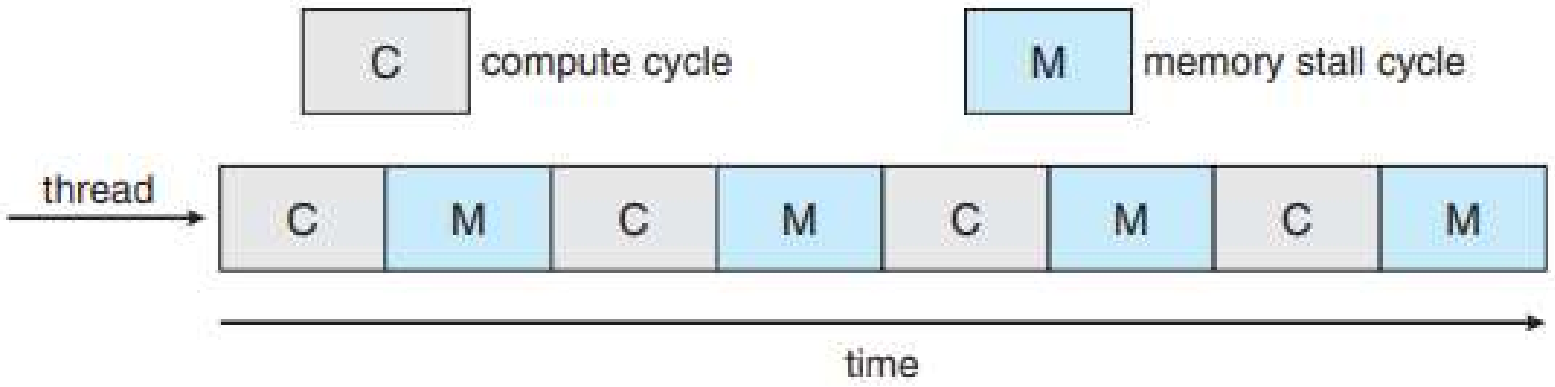
Organization of ready queues



Multicore Processors

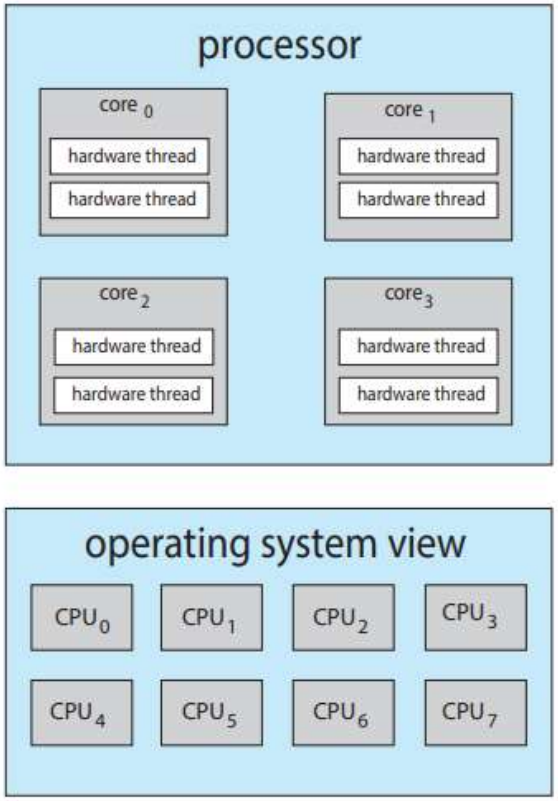
- SMP systems allow several processes to run in parallel by providing multiple physical processors.
- Multicore processors – placing multiple computing cores on the same physical chip.
  - Each core maintains its architectural state and for OS it appears to be separate logical CPU.
  - SMPs that use multicore processors are faster and consume less power than its own physical chip.
  - Memory stalls -
    - Significant delay for data accesses
    - Cache miss (accessing data that are not in cache memory)

Multicore Processors – Memory stall



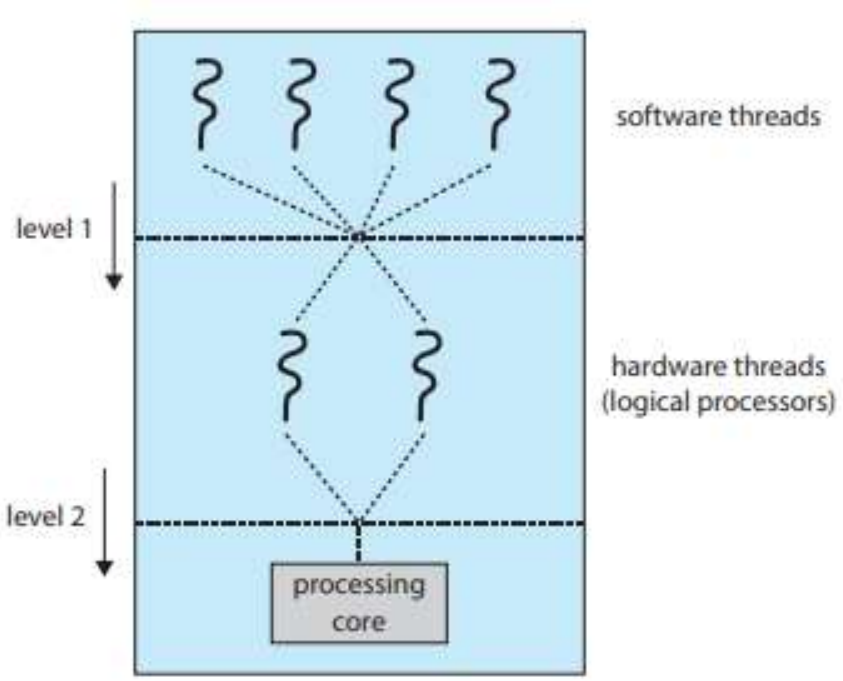
- Remedies
  - Chip multithreading- Using 2 or more hardware threads assigned for each core - if one hardware thread stalls while waiting for memory, the core can switch to another thread.

Chip multithreading



- Here processor contains 4 computing cores, with each core containing 2 hardware threads.
  - OS perspective 8 logical CPUs.
  - Intel processor uses – hyper-threading / simultaneous multithreading/ SMT
- Ways of multithread a processing core
  - **Coarse grained** - a thread executes on a core until a long-latency event such as a memory stall occurs. Switching cost is high as pipelines must flush to begin new execution.
  - **Fine-grained** – switches between threads at a much finer level – at boundary of an instruction cycle. Cost of switching between threads is small.

Multithreaded multicore processor scheduling



Dual threaded processing core

- Requires 2 different levels of scheduling.
- Level 1- scheduling decisions made by OS as it chooses which software thread to run on each hardware thread (logical CPU).
- Level 2- specifies how each code decides which hardware thread to run.
  - Eg: RR to schedule a HW thread to the processing core.

Load Balancing

- To keep the workload evenly balanced / distributed among all processors.
- Load balancing is necessary on systems where processor has its own private ready queues.
- Systems with common ready queues, LB is unnecessary - because once a processor becomes idle, it immediately extracts a runnable thread from the common ready queue.
- General approaches
  - **Push migration** - periodically checks the load on each processor and—if it finds an imbalance—evenly distributes the load by moving (or pushing) threads from overloaded to idle or less-busy processors.
  - **Pull migration** - when an idle processor pulls a waiting task from a busy processor.

Processor Affinity

- Warm cache - when thread has been running on a specific processor most recently accessed data is populated in cache. Successive memory accesses are satisfied in cache.
- When thread migrates to another processor due to load balancing, content of cache needs invalidate and must be repopulated (high cost).
- Most OS's with SMPs avoid migrating threads and keep a thread running on the same processor and maintain warm cache.
- Processor Affinity – a process has an affinity for the processor on which is currently running.
  - Soft affinity - attempting to keep a process running on the same processor.
    - sched setaffinity() system call in Linux
  - Hard affinity – allowing process to migrate between processors during load balancing. Allows a process to specify a subset of processors on which it can run.

Heterogeneous Multiprocessing (HMP)

- Some systems are now designed using cores that run the same instruction set yet vary in terms of their clock speed and power management, including the ability to adjust the power consumption of a core to the point of idling the core.

Advantages :

- By combining several slower cores with faster ones, a CPU scheduler can assign tasks that do not require high performance, but may need to run for longer periods, (such as background tasks) to little cores, thereby helping to preserve a battery charge.
- Interactive applications which require more processing power, but may run for shorter durations, can be assigned to big cores.

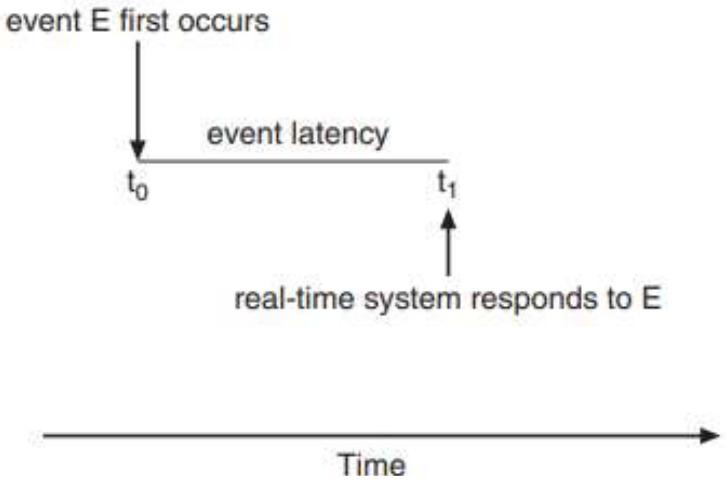
Scheduling algorithms : 9. Real-time CPU Scheduling

- Soft real-time systems
  - provide no guarantee as to when a critical real-time process will be scheduled.
  - Only guarantee that the process will be given preference over non-critical processes.
- Hard real-time systems (strict requirements)
  - A task must be serviced by its deadline; service after the deadline has expired is the same as no service at all

.

Minimizing Latency

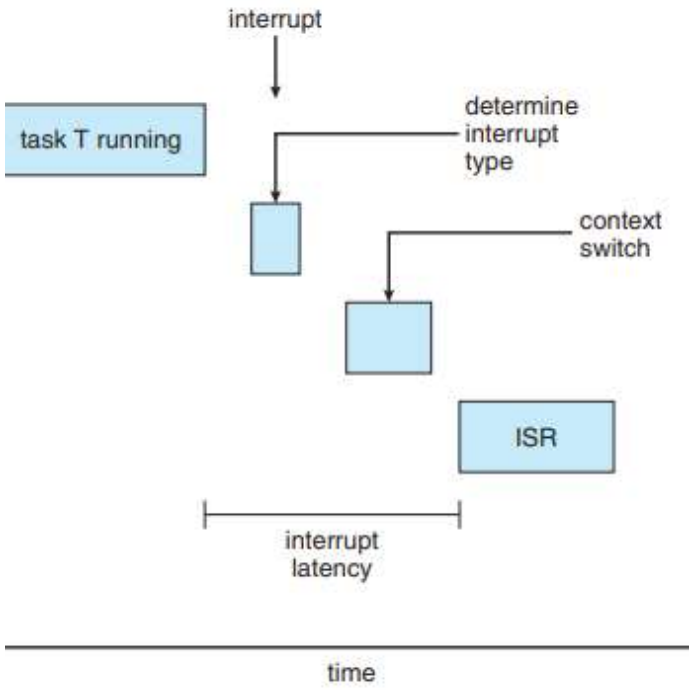
- Eg: Event-driven nature of a real-time system
- System is waiting for an event in real time to occur.
  - Events may arise - In software (as when timer expires) / In hardware
- Upon occurrence of an event – system must respond to and service it as quickly as possible.
- Event latency – amount of time that elapsed from when an event occurs to when it is serviced.



Minimizing Latency

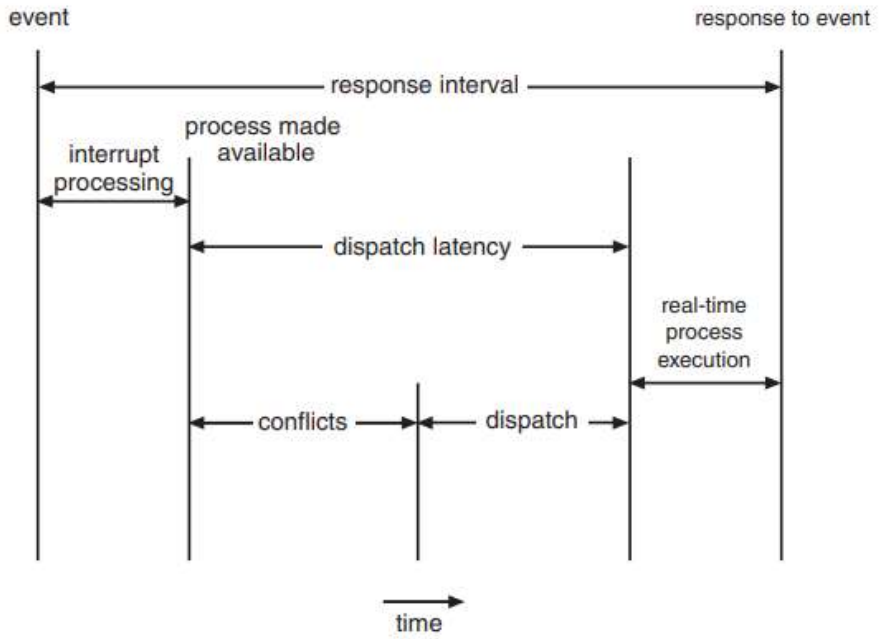
- Types of latencies affect the performance of RT systems.
  - Interrupt latency

  The period of time from the arrival of an interrupt at the CPU to the start of the routine that services the interrupt
    - When an interrupt occurs, the operating system must first complete the instruction it is
    - executing and determine the type of interrupt that occurred.
    - It must then save the state of the current process before servicing the interrupt using the specific
    - interrupt service routine (ISR).
    - The total time required to perform these tasks is interrupt latency.
  - Dispatch latency : amount of time required for the scheduling dispatcher to stop one process and start another



Minimizing Latency

- Dispatch latency
  - Most effective way of keeping dispatch latency low is to provide preemptive kernels.
  - Conflict phase
  1. Preemption of any process running in the kernel
  2. Release by low-priority processes of resources needed by a high-priority process

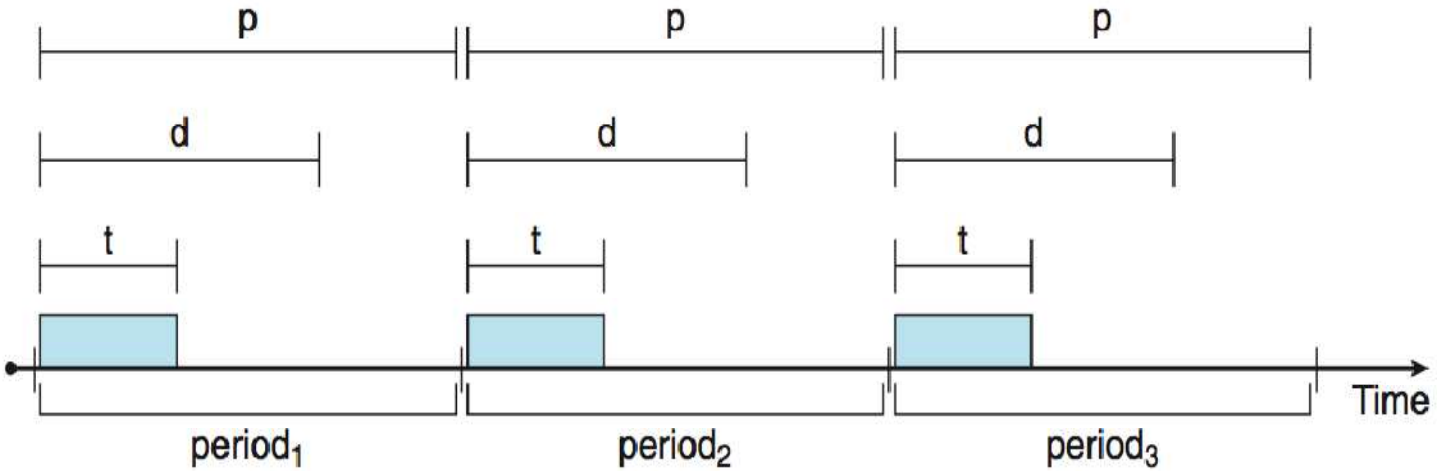


Scheduling algorithms : 10. Priority-based Scheduling

- Real-time operating systems – immediately respond to real-time process as soon as that process requires the CPU.
  - Real OS must support a priority-based algorithm with preemption.
  - Linux, Windows , Solaris OS's support soft real-time scheduling.
    - Windows 32 – has different priority levels. The highest levels—priority values 16 to 31—are reserved for real-time processes.
    - Solaris and Linux have similar prioritization schemes.
- Providing a preemptive, priority-based scheduler only guarantees soft real-time functionality.
- Hard real-time systems must further guarantee that real-time tasks will be serviced in accord with their deadline requirements,

Scheduling algorithms : 10. Priority-based Scheduling (cont'd)

- Processes have new characteristics: **periodic** ones require CPU at constant intervals
  - Has processing time *t*, deadline *d*, period *p*
  - $0 \le t \le d \le p$
  - **Rate** of periodic task is $1/p$



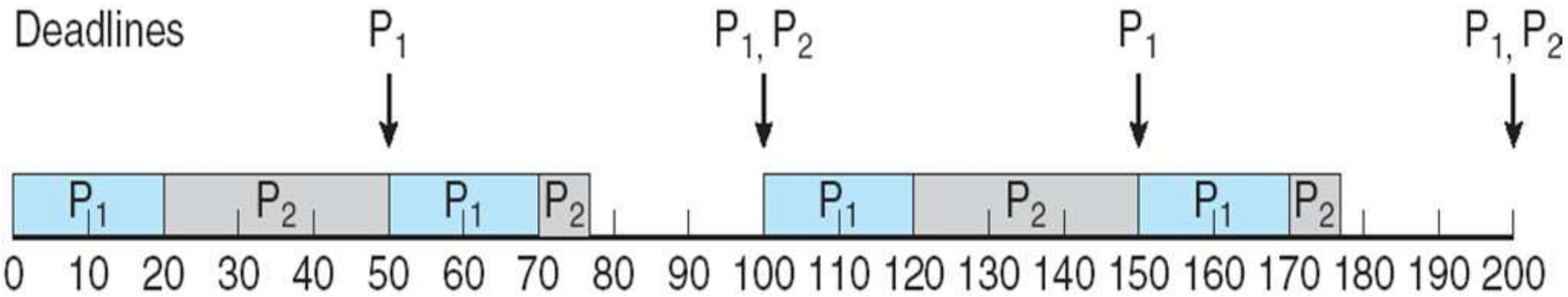
Scheduling algorithms : 10. Priority-based Scheduling (cont'd)

Features

- Process may have to announce its deadline requirements to the scheduler.
- Admission-control algorithm of the scheduler does one of two
- things.
  - It either admits the process, guaranteeing that the process will complete on time, or
  - Rejects the request as impossible if it cannot guarantee that the task will be serviced by its deadline.

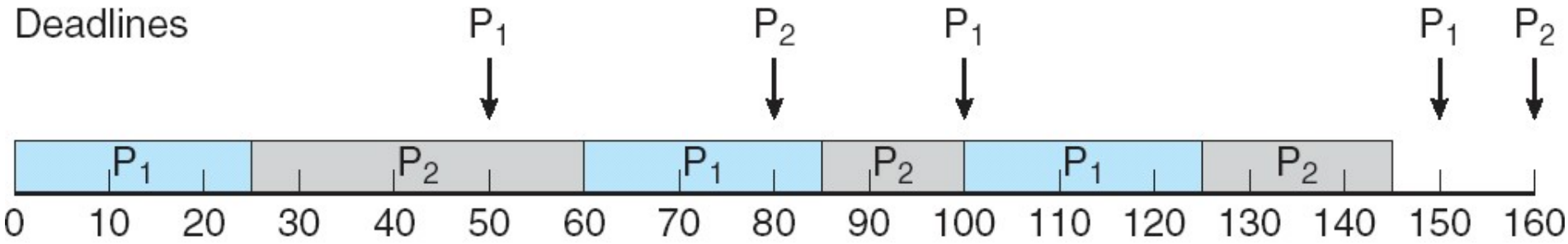
Scheduling algorithms : 11. Rate Monotonic Scheduling

- Schedules periodic tasks using a static priority policy with preemption.
- A priority is assigned based on the inverse of its period
  - Shorter periods = higher priority;
  - Longer periods = lower priority



P_1 (50) and burst t_1 =25 , P_2 (80) and burst t_2=35. P_1 has higher priority due to shorter period.
Utilization (25/50) + (35/80) = 0.94. Could schedule and 6% left as well.

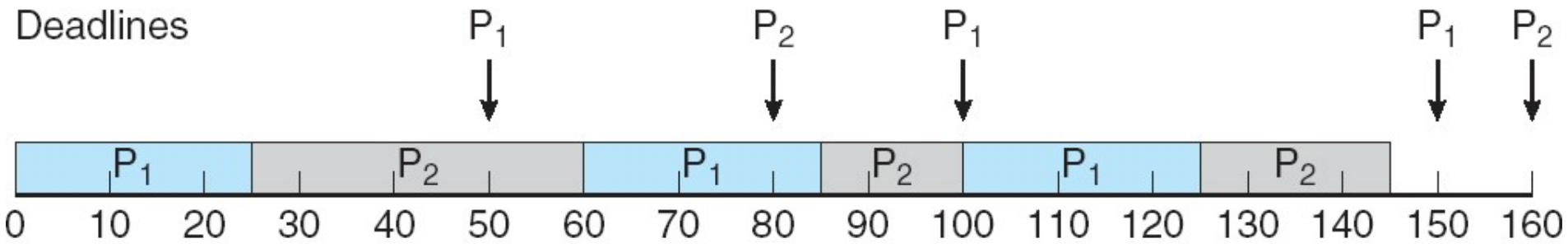
Scheduling algorithms : 11. Rate Monotonic Scheduling



- P_1 (period is 50) and burst t_1 =25 , P_2 (period is 80), and burst t_2 =35
- P1 runs until it completes its CPU burst at time 25.
- Process P2 then begins running and runs until time 50, when it is preempted by P1.
- At this point, P2 still has 10 milliseconds remaining in its CPU burst.
- Process P1 runs until time 75; consequently, P2 finishes its burst at time 85, after the deadline for completion of its CPU burst at time 80.

Scheduling algorithms : 12. Earliest Deadline First Scheduling (EDF)

- Priorities are assigned according to deadlines:
  - The earlier the deadline, the higher the priority
  - The later the deadline, the lower the priority



Linux Scheduling

- Prior to version 2.5 – UNIX scheduling algorithm
  - Not designed with SMP systems.
  - Did not adequately support systems with multiple processors – resulted poor performance for systems with a larger number of processors.
  - With 2.5 – O(1) scheduler
  - Increased support for SMP systems – processor affinity, loading balancing among processors.
    - Excellent performance on SMP systems and Poor performance for interactive processes.
  - Kernel Release 2.6.23 – Completely Fair Scheduler (CFS) became default Linux Scheduling algorithm.
    - Scheduling is based on scheduling classes. Each class has a specific priority. With different classes kernel can accommodate different scheduling algorithms based on needs.

Linux Scheduling

- To decide which task to run next,
- Scheduler selects the highest-priority task belonging to the highest-priority scheduling class.
- Standard Linux kernels implement two scheduling classes:

1. A default scheduling class using the CFS scheduling algorithm
2. A real-time scheduling class

Linux Scheduling

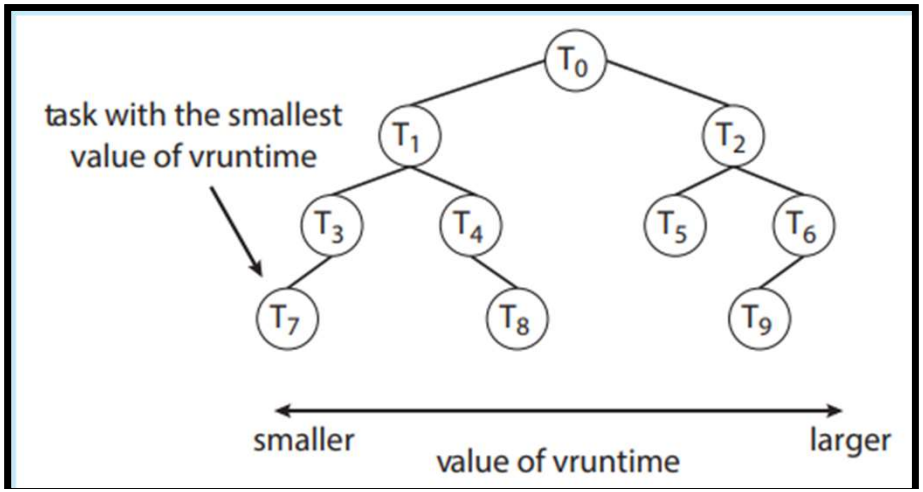
- CFS scheduler assigns a proportion of CPU processing time to each task.
- This proportion is calculated based on the nice value assigned to each task.
- Nice values range from −20 to +19, where a numerically lower nice value indicates a higher relative priority.
- Tasks with lower nice values receive a higher proportion of CPU processing time than tasks with higher nice values. The default nice value is 0.
- CFS doesn't use discrete values of time slices and instead identifies a targeted latency, which is an interval of time during which every runnable task should run at least once.
- CFS scheduler doesn't directly assign priorities. it records how long each task has run by maintaining the virtual run time of each task using the per-task variable **vruntime**.
- The virtual run time is associated with a decay factor based on the priority of a task
  - lower-priority tasks have higher rates of decay than higher-priority tasks.
  - For tasks at normal priority (nice values of 0), virtual run time is identical to actual physical run time.

Linux Scheduling : CFS Scheduler

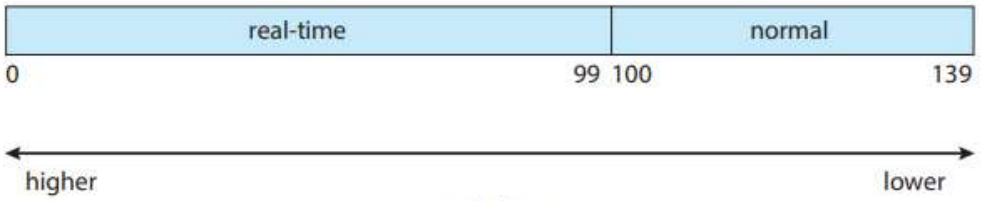
- Assume that two tasks have the same nice values.
- One task is I/O-bound, and the other is CPU-bound. Usually, the I/O-bound task will run only for short periods before blocking for additional I/O, and the CPU-bound task will exhaust its time period whenever it has an opportunity to run on a processor.
- Therefore, the value of **vruntime** will eventually be lower for the I/O-bound task than for the CPU-bound task, giving the I/O-bound task higher priority than the CPU-bound task.
- If the CPU-bound task is executing when the I/O-bound task becomes eligible to run (for example, when I/O the task is waiting for becomes available), the I/O-bound task will preempt the CPU-bound task.

Performance of CFS

- Linux CFS scheduler provides an efficient algorithm for selecting which task to run next.
- Runnable task is placed in a red-black tree.
- As the red-black tree is balanced, navigating it to discover the leftmost node will require O(log N)



Linux Scheduling : RT scheduling



- Uses POSIX standard.
- Tasks are scheduled either e SCHED FIFO or the SCHED RR
- Real-time policy runs at a higher priority than normal (non-real-time) tasks.
- Uses 2 separate priority ranges - one for real-time tasks and a second for normal tasks.
- Real-time tasks are assigned static priorities within the range of 0 to 99, and normal tasks are assigned priorities from 100 to 139.
- These two ranges map into a global priority scheme wherein numerically lower values indicate higher relative priorities. Normal tasks are assigned a priority based on nice values. Where value of −20 maps to priority 100 and a nice value of +19 maps to 139

Windows Scheduling

- Schedules threads using a priority based preemptive scheduling algorithm.
- Dispatcher handles scheduling.
- A thread selected to run by the dispatcher will run until it is preempted by a higher-priority thread, until it terminates, until its time quantum ends, or until it calls a blocking system call, such as for I/O.
- If a higher-priority real-time thread becomes ready while a lower-priority thread is running, the lower-priority thread will be preempted.

Windows Scheduling

- The dispatcher uses a 32-level priority scheme to determine the order of thread execution.
- Priorities are divided into two classes.
  - **Variable class :** contains threads having priorities from 1 to 15,
  - **Real-time class :** contains threads with priorities ranging from 16 to 31.
- The dispatcher uses a queue for each scheduling priority and traverses the set of queues from highest to lowest until it finds a thread that is ready to run.
- If no ready thread is found, the dispatcher will execute a special thread called the idle thread.

Windows Scheduling

Windows API identifies the following six priority classes to which a process can belong

- IDLE PRIORITY CLASS 4
- BELOW NORMAL PRIORITY CLASS 6
- NORMAL PRIORITY CLASS 8
- ABOVE NORMAL PRIORITY CLASS 10
- HIGH PRIORITY CLASS 13
- REALTIME PRIORITY CLASS 24

Relative priorities

- IDLE
- BELOW NORMAL
- NORMAL
- ABOVE NORMAL
- HIGHEST
- TIME CRITICAL

	real-time	high	above normal	normal	below normal	idle priority
time-critical	31	15	15	15	15	15
highest	26	15	12	10	8	6
above normal	25	14	11	9	7	5
normal	24	13	10	8	6	4
below normal	23	12	9	7	5	3
lowest	22	11	8	6	4	2
idle	16	1	1	1	1	1

Thank you!