CAS CS 552
Intro to Operating Systems

Richard West

CPU Scheduling

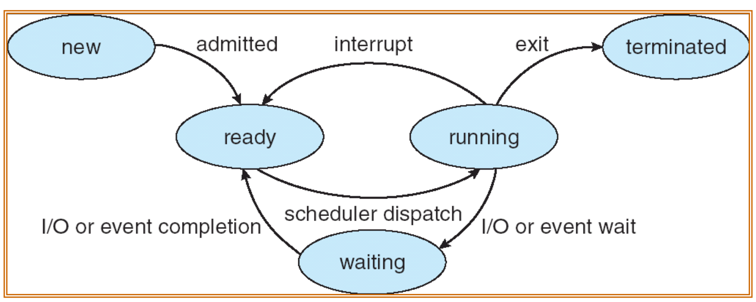
CPU Scheduling

- CPU scheduler (or short-term scheduler) selects one of possibly multiple ready processes/threads to execute on available CPUs
- Processes/threads that are ready for execution are queued in a ready queue
  - When dispatched, a ready process/thread moves into the running state

Preemptive Scheduling 1/2

- CPU scheduling usually occurs under the following conditions:
  - (1) When a process switches from the running to waiting state (e.g., due to a blocking I/O request)
  - (2) When a process switches from the running to ready state (e.g., an interrupt occurs to signify end of current time slice)
  - (3) When a process switches from the waiting to ready state (e.g., upon completion of an I/O request)
  - (4) When a process terminates
  - Other cases? Possibly, when a new process enters the system and is added to the ready queue
  - NOTE: *Preemptive scheduling* apply to cases (2) and (3)
    - Possible to interrupt current process (using CPU) in preference for a new one

Diagram of Process State



Preemptive Scheduling 2/2

- Under non-preemptive scheduling, once a CPU is allocated to a process, that process keeps the CPU until it explicitly relinquishes it, either by terminating or switching to the waiting state
  - Co-routines are a way to voluntarily relinquish the CPU
- Preemptive scheduling occurs when some process is moved from the running to ready state

- A hardware timer is usually needed for preemptive scheduling, to force a trap to the OS so that a (possibly new) process can be allocated the CPU

Dispatcher

- The scheduler <u>selects</u> the next process for execution
- The dispatcher actually gives control of the CPU to the process selected by the scheduler

- The dispatcher must:
  - switch context (between one process/thread and another)
  - switch to user-mode by…
  - Loading the program counter with the next instruction in the code of the user process

Dispatch Latency

- We want to minimize the overhead of:
  - stopping one process/thread
  - switching to another process/thread and,hence,…
  - starting the new process/thread

These overheads define the **dispatch latency**

Scheduling Criteria

- Different performance objectives of processes influence the "most appropriate" choice of scheduling policy/algorithm
  - Performance objectives influence the rules for selecting the next process for execution, and also the data structures used for the "ready queue"

Example Scheduling Criteria/Objectives

- **CPU utilization** – aim to keep this high in most cases, for efficiency
- **Throughput** – number of processes completing execution per unit time
- **Turnaround time** – the interval between when a process 1st arrives (i.e., is created) and when it completes execution
- **Waiting time** – the sum of the periods spent waiting for the CPU (i.e., not executing)
  - Can be time spent in ready queue as well as other wait queues for resources other than CPU
- **Response time** – time between submission of a request and the 1st response of a process

Scheduling Criteria (continued)

- Observe that with response time, we are merely interested in the time to produce the 1st response, but not the time it takes to output that response
  - Outputting a response is limited by the speed of the I/O device, which affects turnaround time
- Other scheduling criteria?
  - What about for real-time systems?

Optimization Criteria

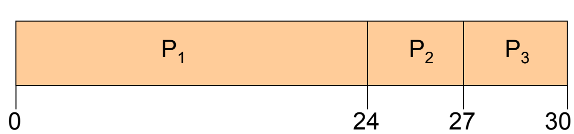
- Max CPU utilization
- Max throughput
- Min turnaround time
- Min waiting time
- Min response time

- Also…
  - Meet all deadlines (hard real-time system)
  - Meet statistical number of deadlines (soft real-time system)
  - Minimize jitter, or delay variation in servicing periodic processes

FCFS Scheduling

Process	Burst Time
P_1	24
P_2	3
P_3	3

- Suppose that the processes arrive in the order: P_1 , P_2 , P_3
  The Gantt Chart for the schedule is:



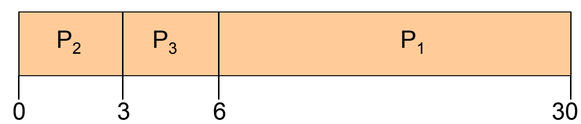
- Waiting time for P_1 = 0; P_2 = 24; P_3 = 27
- Average waiting time: (0 + 24 + 27)/3 = 17

FCFS Scheduling (Cont.)

Suppose that the processes arrive in the order

$$P_2 , P_3 , P_1$$

- The Gantt chart for the schedule is:



- Waiting time for $P_1 = 6; P_2 = 0; P_3 = 3$
- Average waiting time: (6 + 0 + 3)/3 = 3
- Much better than previous case
- **Problem:** processes with short execution (burst) times may have to wait for long-executing processes

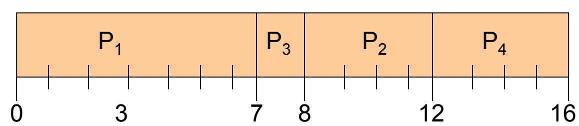
Shortest-Job-First (SJF) Scheduling

- Associate with each process the length of its next CPU burst -- Use these lengths to schedule the process with the shortest time
- Two schemes:
  - nonpreemptive – once CPU given to the process it cannot be preempted until completes its CPU burst
  - preemptive – if a new process arrives with CPU burst length less than remaining time of current executing process, preempt. This scheme is know as the **Shortest-Remaining-Time-First (SRTF)**
- (Preemptive) SJF is optimal w.r.t. **minimizing average waiting time** for a given set of processes

Example of Non-Preemptive SJF

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- SJF (non-preemptive)

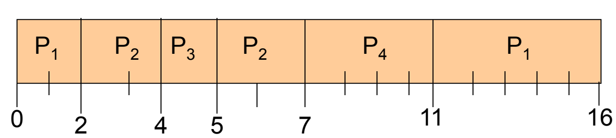


- Average waiting time = (0 + 6 + 3 + 7)/4 = 4

Example of Preemptive SJF

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- SJF (preemptive)



- Average waiting time = (9 + 1 + 0 +2)/4 = 3

SJF Problem

- How do we know the burst time (i.e., actual CPU time) of a process before it executes?

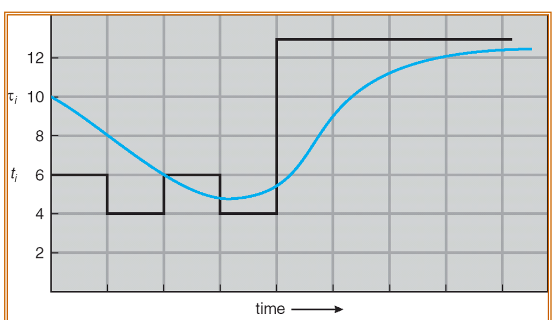
Determining CPU Burst Length

- (1) pre-profile the execution of a process
  - Certain profilers calculate execution time of a process but is this indicative of best/worst/average/actual case?
    - Depends on conditions under which process executes
- (2) Predict next CPU burst length by an exponential weighted average of previous CPU burst lengths

Weighted Average Prediction

- Let t_n be the measured length of the n^{th} burst
- Let τ_{n+1} be the predicted value for the next $(n+1)^{th}$ burst
- Then, for a weighted value, α, where $0 \le \alpha \le 1$ we have
  - $\tau_{n+1} = \alpha t_n + (1 - \alpha)\,\tau_n$
  - Note: τ_n represents the predicted value up to the n^{th} burst

Prediction of the Length of the Next CPU Burst



CPU burst (t_i)		6	4	6	4	13	13	13	…
"guess" (τ_i)	10	8	6	6	5	9	11	12	…

Examples of Exponential Averaging

- $\alpha = 0$
  - $\tau_{n+1} = \tau_n$
  - Recent history does not count
- $\alpha = 1$
  - $\tau_{n+1} = \alpha t_n$
  - Only the actual last CPU burst counts
- If we expand the formula, we get:

$$\begin{aligned}\tau_{n+1} = \alpha\, t_n &+ (1 - \alpha)\alpha t_{n-1} + \ldots \\ &+ (1 - \alpha)^j \alpha\, t_{n-j} + \ldots \\ &+ (1 - \alpha)^{n+1} \tau_0\end{aligned}$$

- Since both α and $(1 - \alpha)$ are less than or equal to 1, each successive term has less weight than its predecessor

Priority Scheduling

- A priority is associated with each process
- The CPU is allocated to the process with the highest priority
  - preemptive
  - nonpreemptive
- SJF is a priority scheduling where priority is based on shortest job
- Problem $\equiv$ Starvation – low priority processes may never execute
- Solution $\equiv$ Aging – as time progresses increase the priority of the process (dynamic priority adjustment)

Round Robin (RR) 1/2

- Designed for time-sharing systems that wish to provide "fair" allocation of CPU resources
- RR is essentially preemptive FCFS
- Each process runs for up to one time quantum (a time slice) before it is preempted
  - A timer interrupt causes a trap to the OS to schedule and dispatch a potentially new process than the one currently using the CPU
  - Preempted processes are put on the back of the ready queue

Round Robin 2/2

- n processes in the ready queue
- Each time quantum = q
- Fairness: each process gets 1/n of the CPU in chunks of size q
- No process waits more than (n-1)q time units before its next quantum of service

- Size of q?
- $q \to \infty$ implies FCFS
- $q \to 0$ implies pure processor sharing (the fluid flow model)
- Small q in practice leads to significant context-switching overhead

Example of RR, q = 20

Process	Burst Time
P_1	53
P_2	17
P_3	68
P_4	24

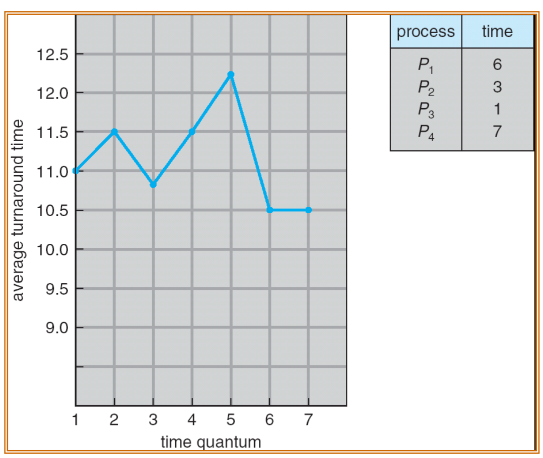
- The Gantt chart is:

P_1	P_2	P_3	P_4	P_1	P_3	P_4	P_1	P_3	P_3

0 20 37 57 77 97 117 121 134 154 162

- Typically, higher average turnaround than SJF, but better *response*

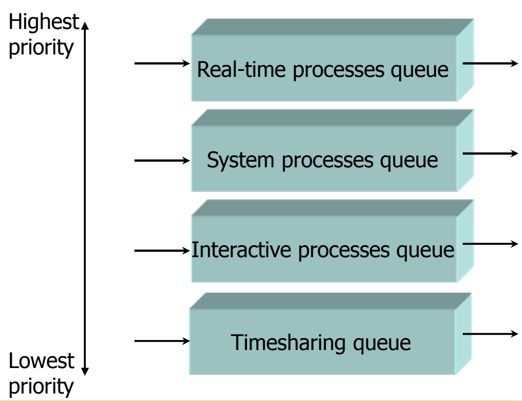
Turnaround Time Varies With The Time Quantum



Multilevel Queue Scheduling 1/2

- Ready queue is partitioned into separate queues
- Each queue has its own scheduling algorithm
  - e.g., "foreground job" queue – RR, background job queue – FCFS
- Scheduling must be done between the queues
  - Typically, fixed priority preemptive scheduling between queues
    - e.g., serve all from foreground then from background
    - Possibility of starvation
  - Alternatively, each queue gets a certain amount of CPU time which it can schedule amongst its processes; e.g., 80% to foreground in RR, 20% to background in FCFS

Example Multilevel Queue Scheduling 2/2



- A timesharing process will only run if there are no processes in the higher priority queues
- Fair queueing, or proportional sharing algorithms, can partition CPU shares amongst job classes

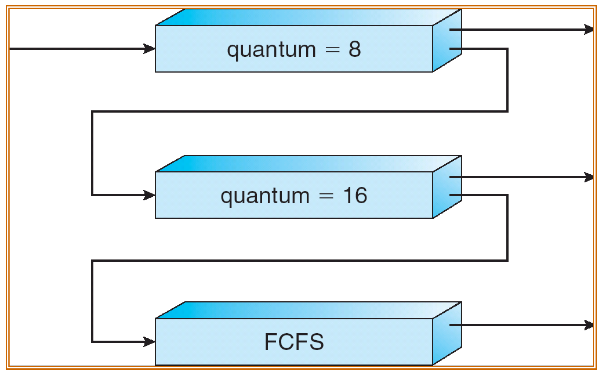
Multilevel Feedback Queue

- A process can move between the various queues; aging can be implemented this way
- 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 which is served FCFS. 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 job is again served FCFS and receives 16 additional milliseconds. If it still does not complete, it is preempted and moved to queue Q_2.

Multilevel Feedback Queues



Only if a higher priority queue is empty will a process in a lower priority queue execute

Multiple-Processor Scheduling

- CPU scheduling more complex when multiple CPUs are available
  - e.g., deadline scheduling on two or more CPUs is NP-hard (Garey & Johnson)

- Various issues come into play that aren't a problem on uniprocessors:
  - Cache affinity
  - Co-runner selection
  - Homogeneous versus heterogeneous processors
  - Load sharing (current load on different CPUs may differ at a given time, t)

Real-Time Scheduling

- Need to meet timeliness constraints
  - Earliest deadline first scheduling (EDF) – essentially a dynamic priority algorithm
  - Rate monotonic scheduling (RM) – essentially a static priority algorithm

  - NOTE: some real-time systems are scheduled "off-line" to guarantee time constraints will be met before process execution

Real-Time Scheduling Problems

- Blocking delays due to resource sharing
- Priority inversion
  - High priority process blocked by a lower priority process
  - Can be solved using e.g., priority inheritance
    - Process executes a "critical section" at priority of highest priority blocked process and then reverts to normal priority after critical section
    - Attempts to avoid "chain blockings" of high priority processes by multiple lower priority processes
  - Priority Ceiling Protocol (PCP) invented to counter deadlock and unresolved chain blocking problem of traditional priority inheritance approaches
  - See the MARS Rover story from 1997!

Scheduling Criteria for RT Systems

- **Lateness** – difference between completion time and deadline of a process
  - "late" if lateness is positive, else "early"
- **Tardiness** – max(0, lateness)
  - i.e., amount by which a process is "late"

Earliest Deadline First

- On a single processor, EDF minimizes maximum lateness (and tardiness)
- Can be shown that "if all deadlines can be met using some ordering of processes, EDF will guarantee to meet all deadlines"
  - Implies least upper bound on resource utilization of 100% (optimal)
  - Can be proven by taking pairs of adjacent processes in a scheduling sequence and swapping them so that the process with a later deadline comes first; under such a situation it can be shown that the lateness is never less than keeping the processes in EDF order
- BUT, what about EDF with overload?!

Least Slack Time Scheduling

- Slack time = time by which a process can be delayed before it will miss its deadline

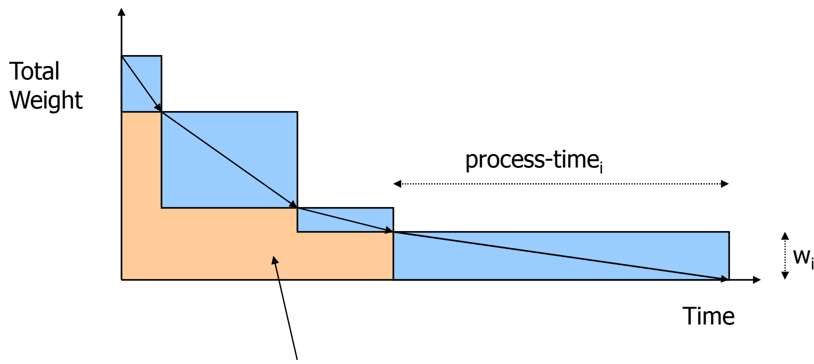
- Least slack time scheduling maximizes minimum process lateness (and tardiness)
  - Not clear how valuable this really is

Static Priority Scheduling

- Minimizes weighted mean delay (, equivalently, weighted mean turnaround time)

- Let priorities = weight/process-time
  - (Remember: SJF minimized mean waiting time)
  - If static priority is calculated as:
    - $weight_i$/process-$time_i$, for each process, P_i, then static priority scheduling minimizes the mean weighted waiting time, $W_{avg}=\Sigma_{\forall i}w_iW_i/\Sigma_{\forall i}w_i$, where w_i=$weight_i$

Static Priority Scheduling



$\Sigma_{\forall i} w_i W_i$ – minimized by lining vectors in a concave manner

Rate Monotonic Scheduling (RMS)

- Highest (static) priority = smallest period for periodic processes
- RM scheduling is a form of static priority preemptive scheduling
- Can guarantee $\Sigma_{i=1 \ldots n} C_i/T_i \leq n(2^{1/n} - 1)$
  - Lim $n \rightarrow \infty$ $\Sigma_{i=1 \ldots n} C_i/T_i = \log_e 2$ (or 69%)

RMS – Concepts and Definitions

- Periodic tasks (a.k.a. jobs/processes/threads)
  - Initiated at fixed intervals
  - Must finish before start of next cycle
  - Task τ_i has utilization, $U_i = C_i/T_i$
    - C_i = computation time
    - T_i = period
    - Total utilization = $U = U_1 + U_2 + \ldots + U_n$

RMS Example

- τ_1: $C_1=40$, $T_1=100$, $U_1=0.4$
- τ_2: $C_2=40$, $T_2=150$, $U_2=0.267$
- τ_3: $C_3=100$, $T_3=350$, $U_3=0.286$

- Utilization of 1st two tasks is $0.4+0.267=0.667 < U(n=2) = 0.828$

- $U = 0.4+0.267+0.286 = 0.953 > U(n=3) = 0.779$
  - Does this mean the task set is not schedulable?

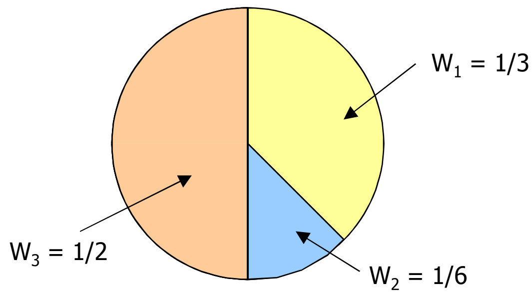
RMS Completion Time Test

- Look at how lower priority tasks are affected by higher priority tasks and see if it's still possible to schedule a task to completion in its current period
- For each task τ_i look at delay due to each higher priority task, τ_j, where $j < i$:
- $W_i(k) = C_i + \sum_{j<i} \lceil W_i(k\text{-}1)/T_j \rceil C_j$
  - $W_i(k)$ = completion time of τ_i after kth iteration of completion time test
  - $W_i(0) = 0$
  - If $W_i(k+1) = W_i(k)$ stop and check that $W_i(k) <= T_i$

Completion Time Test Example

- Using past example:
- $W_3(1) = C_3 + \sum_{j<3} \lceil 0/T_j \rceil C_j = C_3 = 100$
- $W_3(2) = C_3 + \sum_{j<3} \lceil 100/T_j \rceil C_j = 100 + \lceil 100/100 \rceil (40) + \lceil 100/150 \rceil (40) = 180$
- $W_3(3) = 100 + \lceil 180/100 \rceil (40) + \lceil 180/150 \rceil (40) = 260$
- $W_3(4) = 100 + \lceil 260/100 \rceil (40) + \lceil 260/150 \rceil (40) = 300$
- $W_3(5) = 100 + \lceil 300/100 \rceil (40) + \lceil 300/150 \rceil (40) = 300 \Rightarrow$ Done!
- $W_3 = 300 < T_3$ so τ_3 is schedulable using completion time test

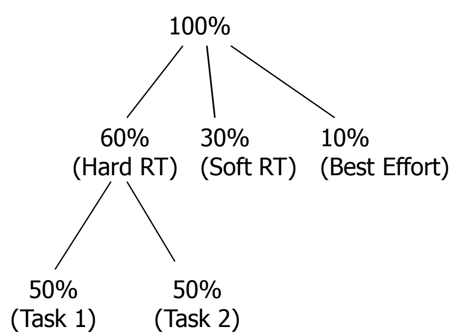
Proportional Share Scheduling



- Allocate resources in proportion to weights

Hierarchical Scheduling

- Extend proportional allocation to a hierarchy



Proportional Sharing

- Aim to model generalized processor sharing
  - Give the illusion of all processes making progress simultaneously, on a hypothetical dedicated processor per process (even when only one physical CPU)

- (Fractional) share, $f_i(t)$, of process, p_i at time t:
  - $f_i(t) = w_i / (\Sigma_{\forall j} w_j)$

Proportional Sharing

- Ideal time using resource (e.g., CPU) in interval $[t_0,t_1]$:

$$S_i(t_o, t_1) = \int_{t_o}^{t_1} f_i(\tau) d\tau$$

- If $s_i(t_0,t_1)$ is the *actual* service time in interval $[t_0,t_1]$, then:

$$lag_i(t_1) = S_i(t_o, t_1) - s_i(t_0, t_1)$$

Proportional Sharing

- It follows that:

$$S_i(t_1,t_2) = w_i \int_{t_1}^{t_2} \frac{1}{\sum_{\forall j} w_j} d\tau$$

- Now, let's define a *virtual* time, V(t):

$$V(t) = \int_0^t \frac{1}{\sum_{\forall j} w_j} d\tau$$

Proportional Sharing

- Therefore,
  - $S_i(t_1,t_2) = (V(t_2) - V(t_1))w_i$
- In the EEVDF (Earliest Eligible Virtual Deadline First) algorithm, a process is eligible for service at time e, when its ideal and actual service times correspond:
  - i.e., $s_i(t_0^i,t) = S_i(t_0^i,e)$
  - NB: t_0^i is time Process P_i becomes active

Proportional Sharing

- $V(e) = V(t_0^i) + s_i(t_0^i, t)/w_i$
  - Virtual eligible time

- $V(d) = V(e) + r/w_i$
  - Virtual deadline at some real-time d, is based on value r:
  - $r = S_i(e,d)$ and represents the service time a new request should receive in the interval [e,d]
    - r is the service time of a request (e.g., a quantum)

Algorithm Behavior

- For each task, iteratively compute:
  - $ve^{(1)} = V(t_0^i)$
  - $vd^{(k)} = ve^{(k)} + r^{(k)}/w_i$
  - $ve^{(k+1)} = vd^{(k)}$

  - Above steps computed for each *serviced request*
  - $ve^{(1)}$ is the virtual eligibility time of the 1^{st} request
  - $ve^{(k)}$ is the virtual eligibility time of the k^{th} request
  - $vd^{(k)}$ is the virtual deadline of the k^{th} request
  - $r^{(k)}$ is the service time of the k^{th} request

Example –2 CPUs 3 Tasks

	Task 1 (τ1)				Task 2 (τ2)				Task 3 (τ3)				Schedule	
t	W1	Ve	Vd	r	W2	Ve	Vd	r	W3	Ve	Vd	r	CPU1	CPU2
0	1	0	1	1	2	0	0.5	1	3	0	0.33	1	τ3	τ2
1	1	0	1	1	2	0.5	1	1	3	0.33	0.67	1	τ3	τ1
2	1	1	2	1	2	0.5	1	1	3	0.67	1	1	τ3	τ2
3	1	1	2	1	2	1	1.5	1	3	1	1.33	1	τ3	τ2
4	1	1	2	1	2	1.5	2	1	3	1.33	1.67	1	τ3	τ1
5	1	2	3	1	2	1.5	2	1	3	1.67	2	1	τ3	τ2
6	1	2	3	1	2	2	2.5	1	3	2	2.33	1	τ3	τ2
7	1	2	3	1	2	2.5	3	1	3	2.33	2.67	1	τ3	τ1
8	1	3	4	1	2	2.5	3	1	3	2.67	3	1	τ3	τ2
9	1	3	4	1	2	3	3.5	1	3	3	3.33	1	τ3	τ2
10	1	3	4	1	2	3.5	4	1	3	3.33	3.67	1	τ3	τ1
11	1	4	5	1	2	3.5	4	1	3	3.67	4	1	τ3	τ2
12	1	4	5	1	2	4	4.5	1	3	4	4.33	1	τ3	τ2
13	1	4	5	1	2	4.5	5	1	3	4.33	4.67	1	τ3	τ1
14	1	5	6	1	2	4.5	5	1	3	4.67	5	1	τ3	τ2
15	1	5	6	1	2	5	5.5	1	3	5	5.33	1	τ3	τ2
16	1	5	6	1	2	5.5	6	1	3	5.33	5.67	1	τ3	τ1
17	1	6	7	1	2	5.5	6	1	3	5.67	6	1	τ3	τ2
18	1	6	7	1	2	6	6.5	1	3	6	6.33	1	τ3	τ2
19	1	6	7	1	2	6.5	7	1	3	6.33	6.67	1	τ3	τ1
20	1	7	8	1	2	6.5	7	1	3	6.67	7	1	τ3	τ2
21	1	7	8	1	2	7	7.5	1	3	7	7.33	1	τ3	τ2
22	1	7	8	1	2	7.5	8	1	3	7.33	7.67	1	τ3	τ1
23	1	8	9	1	2	7.5	8	1	3	7.67	8	1	τ3	τ2