Introduction to Operating Systems
Scheduling

Based on the lecture slides prepared by (1) Joseph Pasquale at UCSD and (2) Zhi Wang at FSU

Outline

- Background: Basics of scheduling
- Xv6 Scheduling
- Multiprocessor/multicore scheduling
- Real-time scheduling

Basic Concepts

- Maximum CPU utilization obtained with multiprogramming
- CPU–I/O Burst Cycle – Process execution consists of a **cycle** of CPU execution and I/O wait
- **CPU burst** followed by **I/O burst**
- CPU burst distribution is of main concern

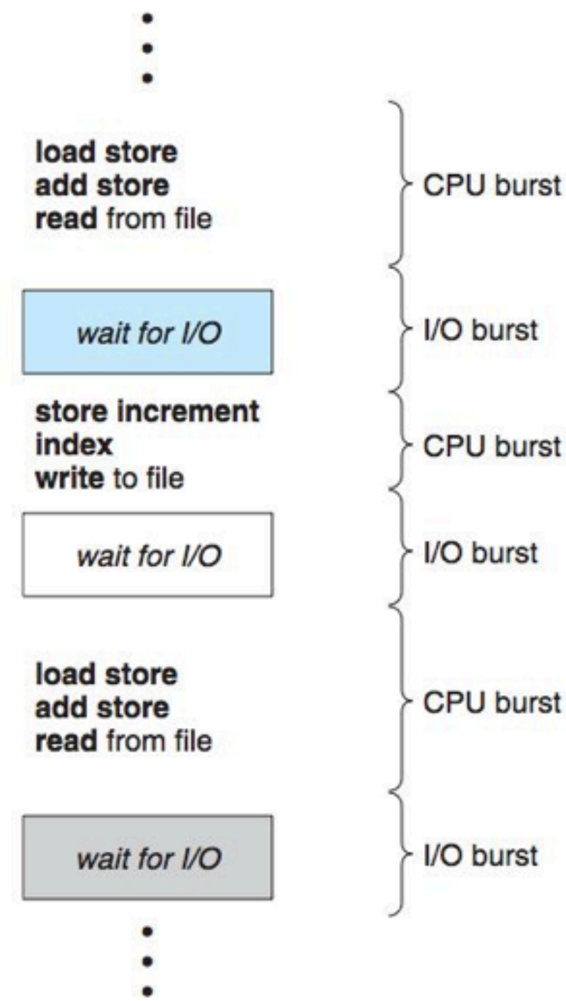


Figure 6.1 Alternating sequence of CPU and I/O bursts.

Histogram of CPU-burst Times

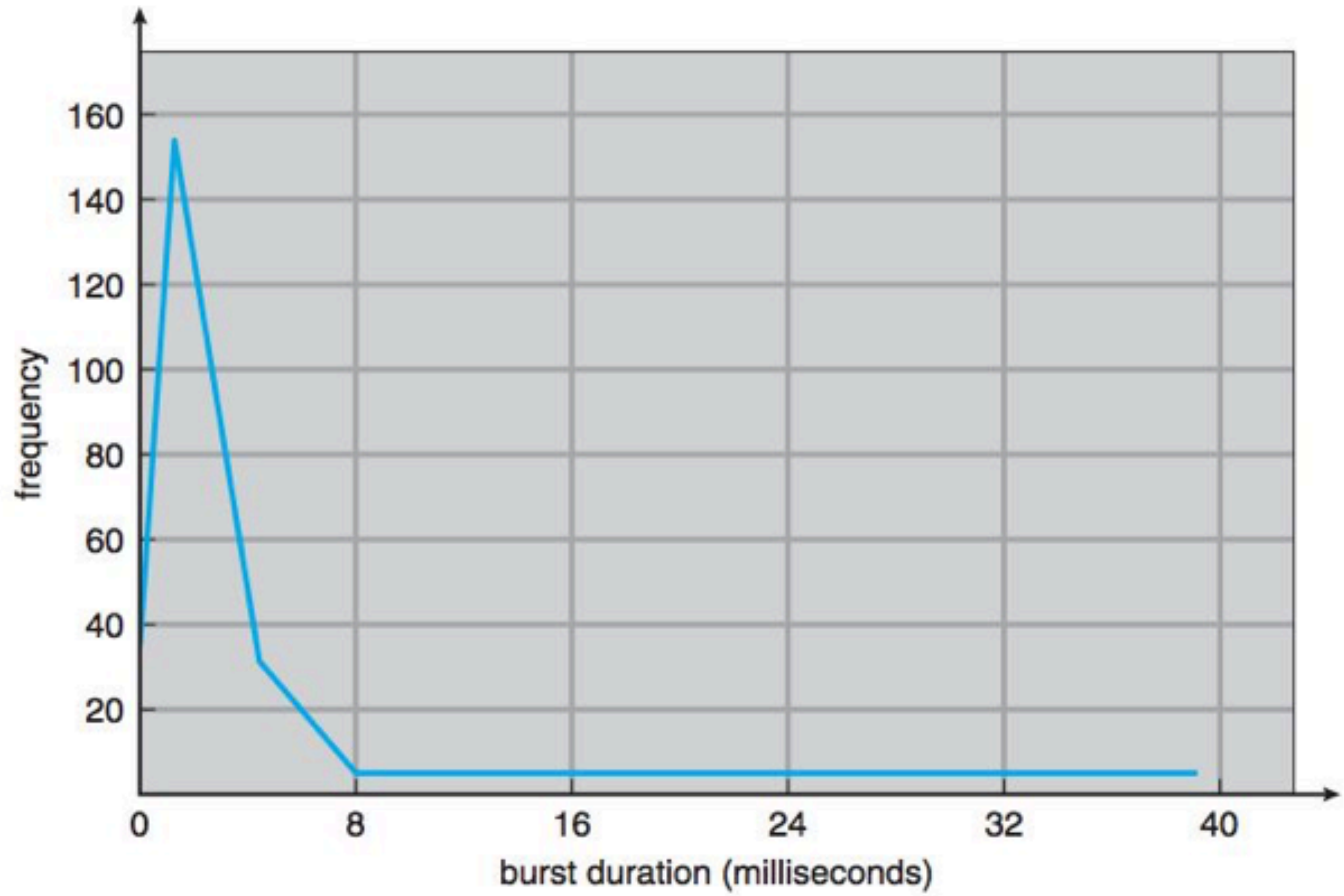


Figure 6.2 Histogram of CPU-burst durations.

The Thread Scheduling Problem

- Given *n* threads and *m* CPUs, $n > m$
  - When a CPU becomes available, which ready thread should be assigned to it?
  - How much time should that thread get to run?

Basics of Scheduling

Thread	Arrival Time	Service Time
A	0	5
B	0	3
C	0	1

- Arrival time: time that thread is created
- Service time: CPU time needed to complete
- Turnaround time: arrival to departure
  - Thread arrives, waits for CPU, then uses (in bursts)
  - Departs after CPU usage equals service time

Scheduling Criteria

- **CPU utilization** – keep the CPU as busy as possible
- **Throughput** – # of processes that complete their execution per time unit
- **Turnaround time** – amount of time to execute a particular process
- **Waiting time** – amount of time a process has been waiting in the ready queue
- **Response time** – amount of time it takes from when a request was submitted until the first response is produced, not output (for time-sharing environment)

Longest First vs. Shortest First

Longest First											
	0	1	2	3	4	5	6	7	8	9	TT
A											5
B											8
C											9
										Average Turnaround Time	**22/3**

Shortest First											
	0	1	2	3	4	5	6	7	8	9	TT
A											9
B											4
C											1
										Average Turnaround Time	**14/3**

Minimize Avg Turnaround Time

- Given n threads with service times S_1, … , S_n
  - Note: threads are numbered 1, 2, 3, …, n
- Average turnaround time computed as follows

Minimize Avg Turnaround Time

- Given n threads with service - mes $S_1, \ldots, S_n$
  - Note: threads are numbered 1, 2, 3, …, n
- Average turnaround - me computed as follows
  - $[S_1 + (S_1 + S_2) + (S_1 + S_2 + S_3) + \ldots + (S_1 + \ldots + S_n)] / n$

Minimize Avg Turnaround Time

- Given n threads with service - mes $S_1, \ldots, S_n$
  - Note: threads are numbered 1, 2, 3, …, n
- Average turnaround - me computed as follows
  - $[S_1 + (S_1 + S_2) + (S_1 + S_2 + S_3) + \ldots + (S_1 + \ldots + S_n)] / n$
  - $[(n \times S_1) + ((n-1) \times S_2) + ((n-2) \times S_3) + \ldots + S_n] / n$

Shortest First Is Provably Optimal

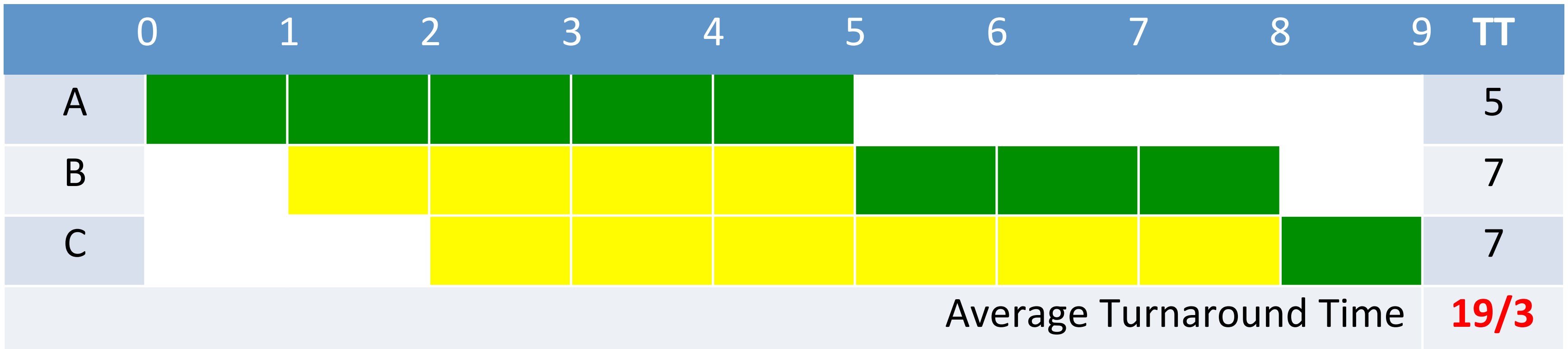
- Given n threads with service - mes $S_1, \ldots, S_n$
  - Note: threads are numbered 1, 2, 3, …, n
- Average turnaround - me computed as follows
  - $[S_1 + (S_1 + S_2) + (S_1 + S_2 + S_3) + \ldots + (S_1 + \ldots + S_n)] / n$
  - $[(n \times S_1) + ((n-1) \times S_2) + ((n-2) \times S_3) + \ldots + S_n] / n$
- In general: order by shortest to longest
  - S_1 has maximum weight (n), minimize it
  - S_2 has next-highest ($n-1$), minimize it after S_1

Consider Different Arrival Times

Thread	Arrival Time	Service Time
A	0	5
B	1	3
C	2	1

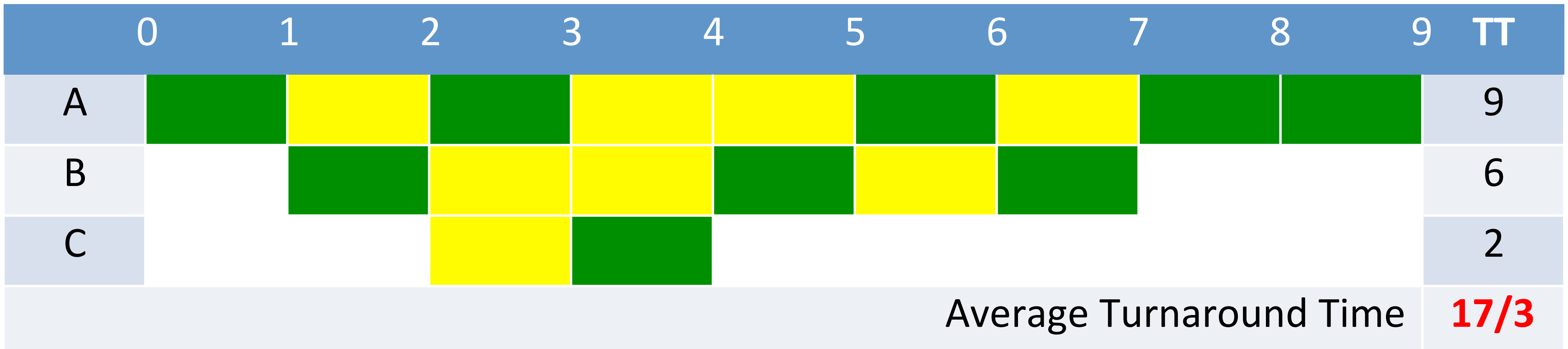
	0	1	2	3	4	5	6	7	8	9	TT
A	*executing*										5
B		*waiting*				*executingg*					7
C			*waiting*						*ex…*		7
									Average Turnaround Time		**19/3**

FCFS: First Come First Served



- Allocate CPU to threads in order of arrival
- A B]A B]A CB]A CB]A CB]A CB]A C]B C]B C]B C
- Average turnaround time = (5 + 7 + 7)/3 = 6.3
- Non-preemptive, simple, no starvation
- Poor for short threads

RR: Round Robin



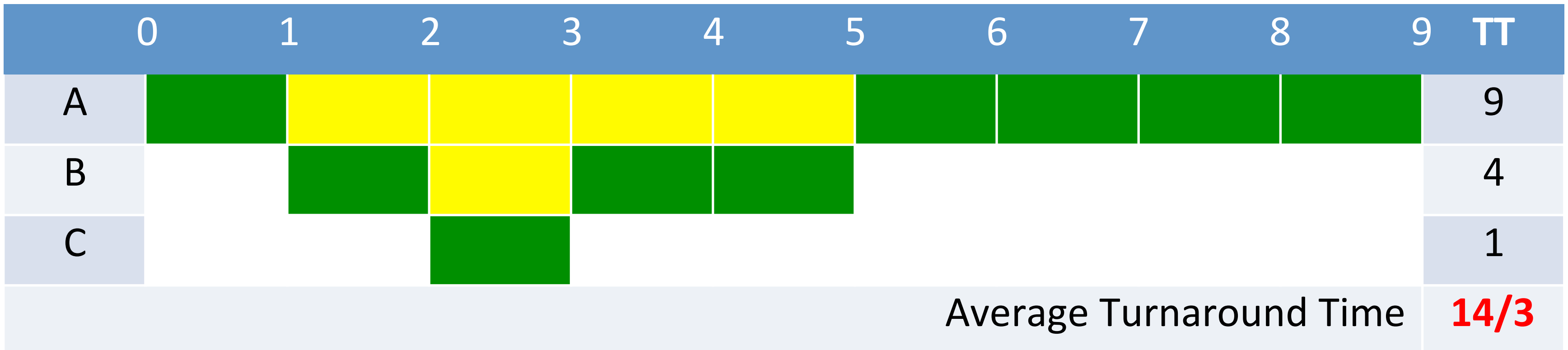
- Time-slice: each thread gets quantum in turn
- A B]A A]B CA]B BC]A AB]C A]B B]A A]B A A
- Average turnaround time = (9 + 6 + 2)/3 = 5.7
- Preemptive, simple, no starvation
- Thread waits at most (*n*–1) x quantum

Shortest Next

	0	1	2	3	4	5	6	7	8	9	TT
A											5
B											8
C											4
										Average Turnaround Time	**17/3**

- Select thread with shortest service - me
- A B]A B]A BC]A BC]A BC]A BC]A B]C B B B
- Average turnaround time = (5 + 8 + 4)/3 = 5.7
- Optimal for non-premptive, allows starvation
- Assumes service times are known

SRT: Shortest Remaining Time



- Select thread with shortest remaining - me
- A B]A A]B AC]B AB]C A]B A]B A A A A
- Average turnaround time = (9 + 4 + 1)/3 = 4.7
- Assumes service times are known
- Optimal for preemptive, but allows starvation

Determining Length of Next CPU Burst

- Can only estimate the length – should be similar to the previous one
  - Then pick process with shortest predicted next CPU burst

- Can be done by using the length of previous CPU bursts, using exponential averaging

1. $t_n =$ actual length of n^{th} CPU burst
2. $\tau_{n+1} =$ predicted value for the next CPU burst
3. $\alpha, 0 \leq \alpha \leq 1$
4. Define: $\tau_{n=1} = \alpha\, t_n + (1-\alpha)\tau_n.$

- Commonly, α set to ½

Determining Length of Next CPU Burst

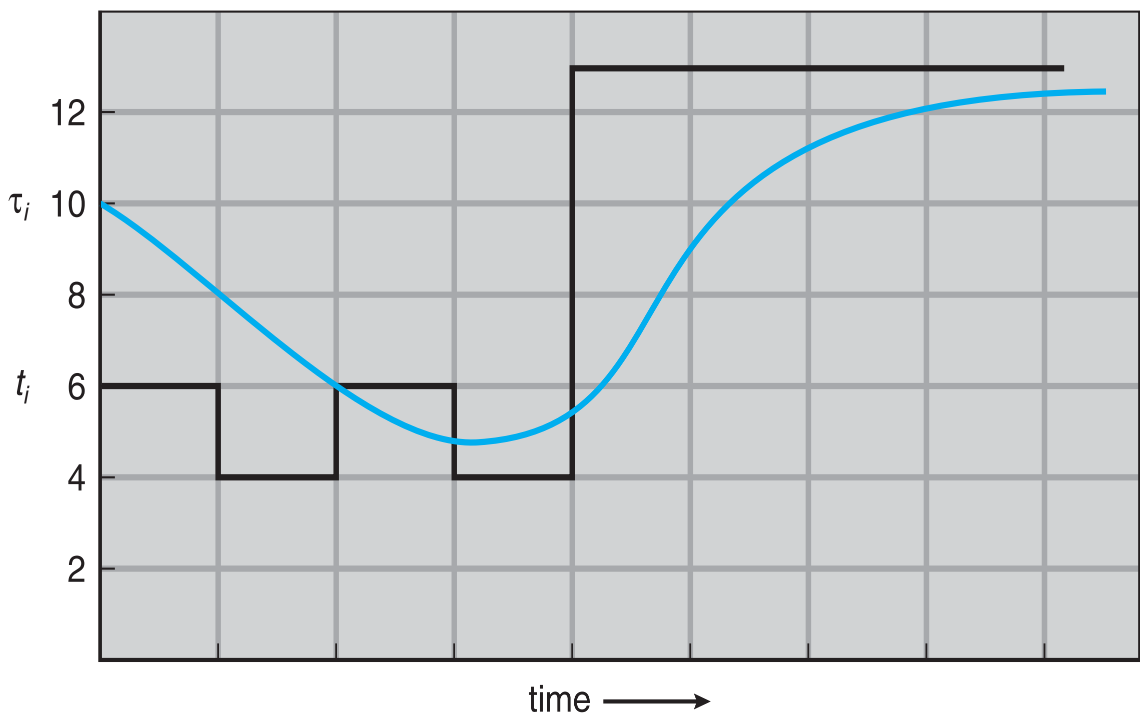
- Can only estimate the length – should be similar to the previous one
  - Then pick process with shortest predicted next CPU burst

- Can be done by using the length of previous CPU bursts, using exponential averaging

  1. $t_n =$ actual length of n^{th} CPU burst
  2. $\tau_{n+1} =$ predicted value for the next CPU burst
  3. $\alpha, 0 \le \alpha \le 1$
  4. Define: $\tau_{n=1} = \alpha\, t_n + (1-\alpha)\tau_n.$

- Commonly, α set to ½

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	…

Xv6 Scheduling

- Xv6 supports preemptive scheduling
  - Process waiting for I/O or a child to exit, or waiting in sleep
  - A timer periodically forces a context switch (freq = 100 ticks/s)
- Xv6 implements a round-robin scheduler

XV6 Scheduler

- Each CPU has its own scheduler kernel thread
  - scheduler() defined in proc.c
  - Each CPU executes scheduler after initialization
    - Initialization happens in mpmain(), which is defined in main.c
- Context switches in Xv6 are performed in two steps
  - Current process → scheduler (in sched() of proc.c)
    - A process' kernel thread switches to the current CPU's scheduler thread
  - Scheduler → the next process (in schedule())
- Context switch is implemented in swtch.S
  - ```
    void swtch(struct context **old, struct
                    context *new);
    ```
    - struct context is defined in proc.h
  - Saves old process' registers in old context
  - Restores new process' registers from new context
    ```

XV6 Scheduler

- scheduler() loops through the ptable to select the next process
  - A round-robin scheduling algorithm
- It calls swtch to switch to the kernel thread of the next process
  - **Kernel stack** and the **registers** are saved and restored
  - swtch returns when the running process calls sched() to switch to the scheduler
- A process can call yield() to voluntarily give up CPU
  - A running process is forced to yield to other processes in the timer interrupt handler for preemptive scheduling (trap.c)

Multi-Level Feedback Queues

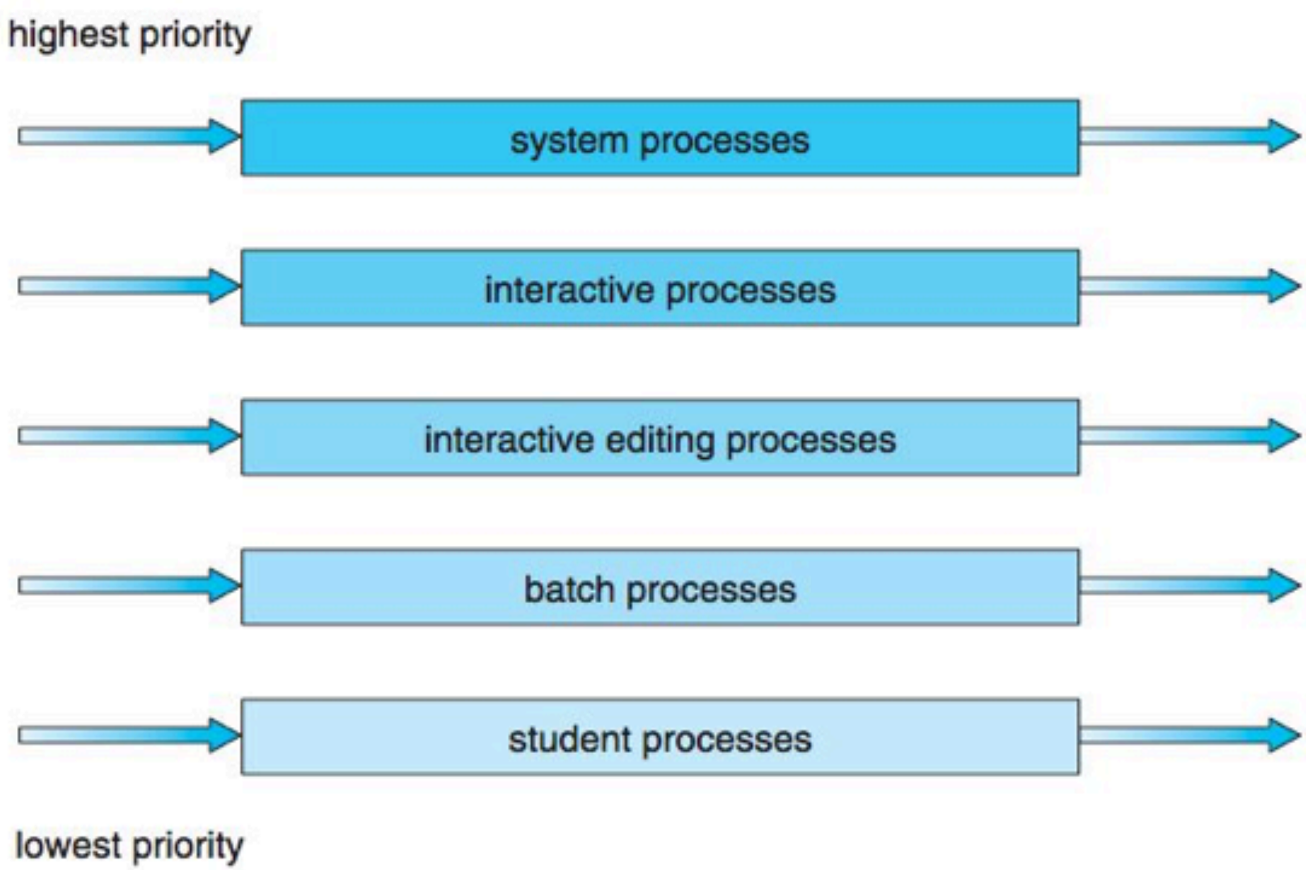
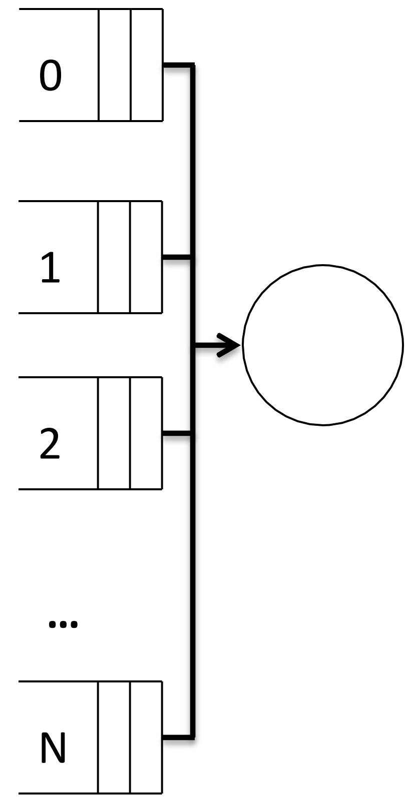


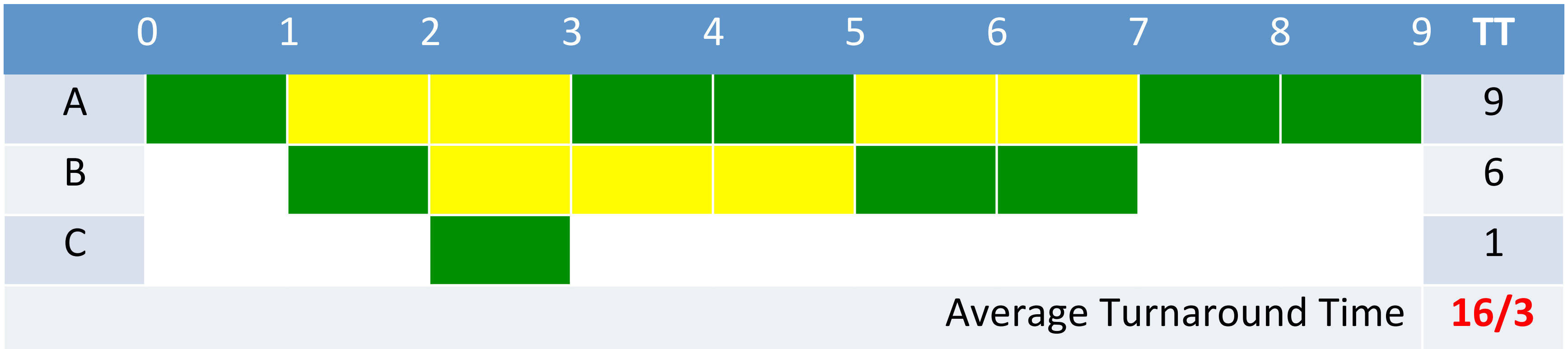
Figure 6.6 Multilevel queue scheduling.

Multi-Level Feedback Queues

- Priority queues 0 (high), …, N (low)
- New threads enter queue 0
- Select from highest priority queue *k*
- Run for 2^k quantums
- Move to next lower prio queue $k+1 \le N$
- If preempted, back to same queue
- Unblocked threads to queue 0

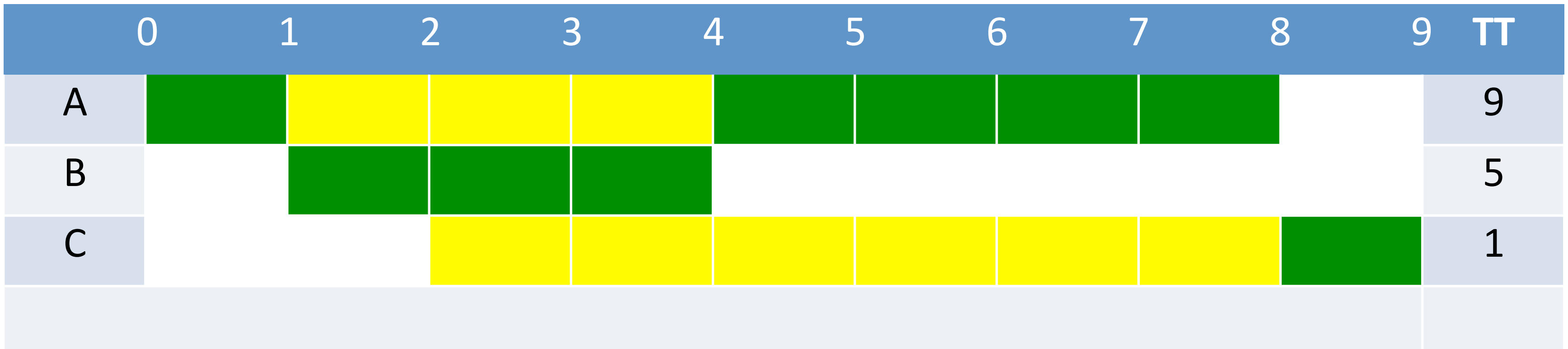


Multi-Level Feedback Queues



- Select from highest queue *k*, run 2^k quantums
- A $B]_0A$ $A]_1B$ $A]_1C]_0B$ $BA]_1C$ $B]_1A(2)$ $A]_2B(2)$ A
- Average Turnaround Time = (9 + 6 + 1)/3 = 5.3
- Complex, adaptive, highly responsive
- Favors shorter over longer, allows starvation

Priority Scheduling



- Select thread with highest priority
  - Example: A_M = medium, B_H = high, C_L = low
- A_M B_H]A_M A_M]B_H $C_L A_M$]B_H $C_L A_M$]B_H(2) C_L]A_M(4) C
- Allows scheduling based on “external” criteria
  - E.g., priority = 1/CPU_time_used

Stride Scheduling (Proportional Share Scheduling)

- For threads A, B, C … with *requests* R_A, R_B, R_C …
- Calculate ***strides***: $S_A = 1/R_A$, $S_B = 1/R_B$, $S_C = 1/R_C$ …
- For each thread *x*, maintain ***pass*** value P_x (init 0)
- Schedule: repeat every quantum
  - Select thread *x* with minimum pass value P_x, run
  - Increment pass value by stride value: $P_x = P_x + S_x$
- Optimization: use only integers for R_x, S_x and P_x
  - Calculate $S_x = L/R_x$ using very large L, e.g., L = 100000

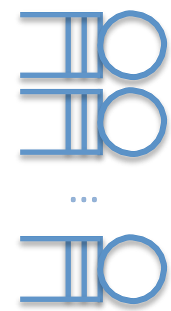
Stride Scheduling Example

Thread x	Runs Now	A	B	C	To Run Next
Requested Utilization: R_x (in %)		50	10	40	
Stride: $S_x = L/R_x$ ($L = 100000$)		2000	10000	2500	
Pass: $P_x = P_x + S_x$; init = 0		0	0	0	A
quantum 1	A	2000	0	0	B
quantum 2	B	2000	10000	0	C
quantum 3	C	2000	10000	2500	A
quantum 4	A	4000	10000	2500	C
quantum 5	C	4000	10000	5000	A
quantum 6	A	6000	10000	5000	C
quantum 7	C	6000	10000	7500	A
quantum 8	A	8000	10000	7500	C
quantum 9	C	8000	10000	10000	A
quantum 10	A	10000	10000	10000	… will repeat

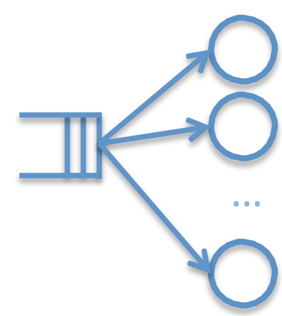
- In 10 quantums, A ran 5; B ran 1; C ran 4

Multiple CPUs/Cores

- Local queue per CPU
  - each CPU has its own queue

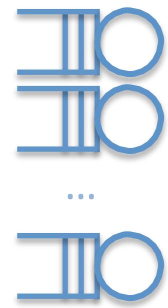


- Single shared Global queue
  - feeds all CPUs

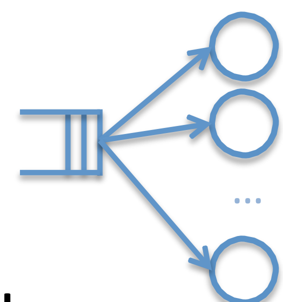


Multiple CPUs/Cores

- Local queue per CPU
  - each CPU has its own queue
  - can lead to imbalance

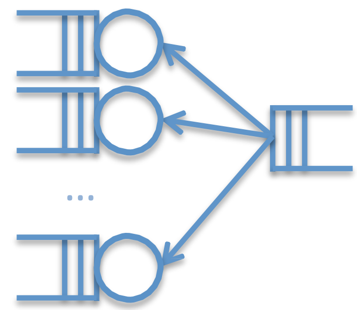


- Single shared Global queue
  - feeds all CPUs
  - accessing queue can be bos leneck



Local Queues + Global Queue

- Local Queue per CPU
  - each CPU has its own queue
- Single shared Global queue
  - feeds all CPUs



- Local queues have priority over Global queue
- When local queue is empty, CPU can still be given work to do

Thread Scheduling Strategies

- Static assignment (queue per CPU)
  - Thread always assigned to the same CPU
- Dynamic assignment (single shared queue)
  - Thread can be assigned to any CPU
- Dynamic load balancing (queue per CPU)
  - Threads can move between queues
  - Avoid queue imbalances (empty while others > 1)
  - Linux uses this approach

What about Kernel threads?

- Master/slave approach
  - Kernel threads always run on same CPU (master)
- Peer approach
  - Any thread, kernel or user, can run on any CPU

Master/Slave Approach

- Key kernel threads always run on master
- Pros
  - Simple, easy to extend uniprocessor-based kernels
- Cons
  - What if master becomes bos leneck
  - What if master fails?

Peer Approach

- Each CPU does its own “self-scheduling”
- Pros
  - Load balancing
  - Fault tolerant
- Cons
  - Complexity
  - Must be careful about synchronization

Timeshare/Multiprogram CPUs?

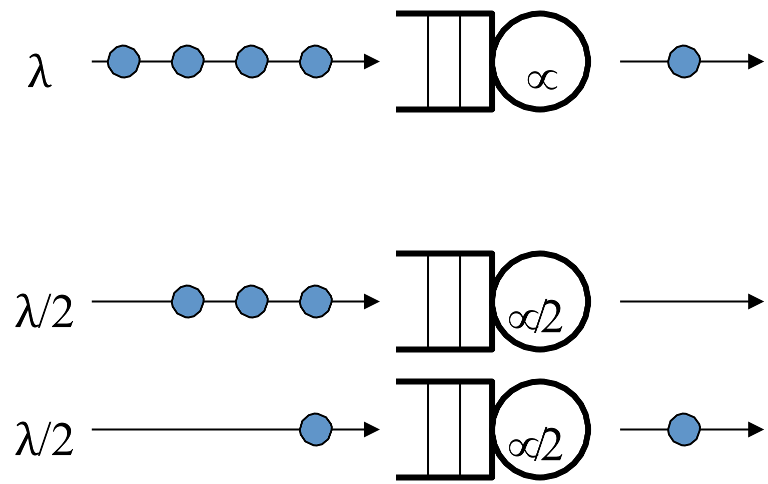
- If there are a large number of CPUs
  - Keep thread on CPU, even if it blocks, until done
- Can lead to high performance
  - Consider a group of related threads
  - Low overhead: no context switching
- Can be "wasteful" – but waste may be OK

Thread Dispatching

- When CPU becomes available, *which* thread?
- For uniprocessors
  - Thread selection: big effect system performance
  - Leads to complex scheduling algorithms
- For system with large number of CPUs/cores
  - Can afford to waste: use simpler algorithms
  - Simple algorithms, less overhead
  - Can improve process multi-threaded performance

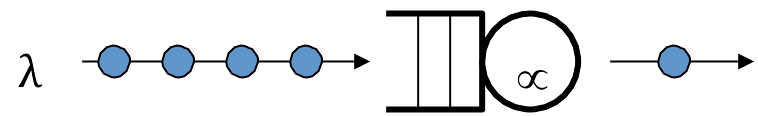
Which is better?

- Single fast CPU with single queue
- Multiple slower CPUs with separate queues

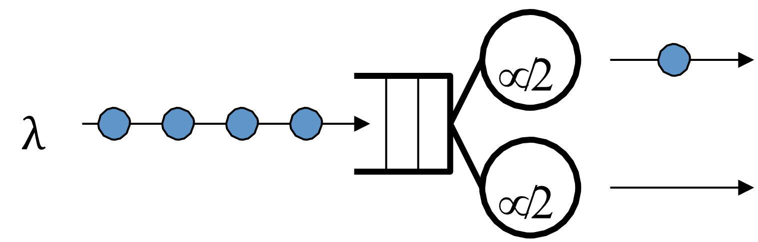


OK, What about This?

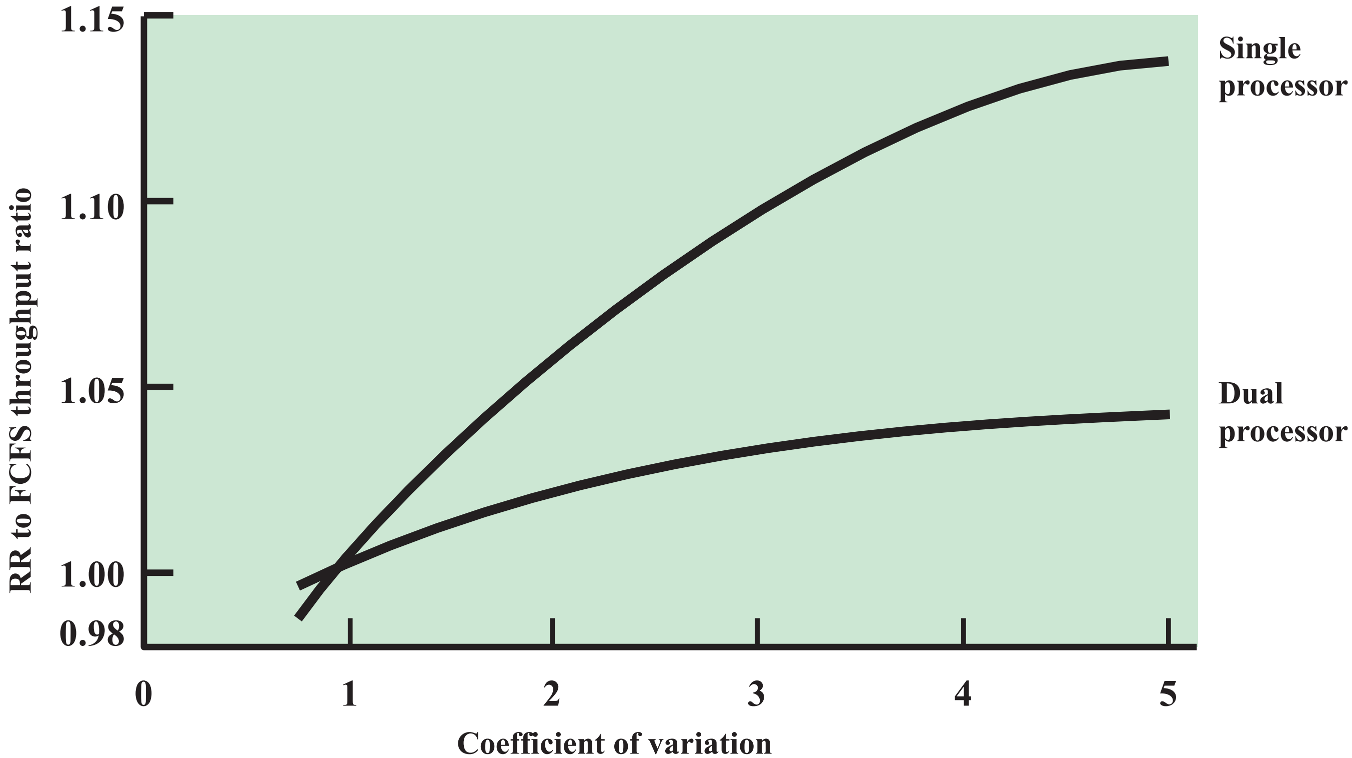
- Single fast CPU with single queue



- Multiple slower CPUs, single queue



RR vs FCFS on Single vs Dual



From Stallings, Opera - n g Systems, 8th Ed., Pearson Publishing, 2015, quo- ng study in Sauer and Chandy, Computer Systems Performance Modeling, Prentice Hall, 1981.

Gang Scheduling

- Schedule a *group* of cooperating threads to run simultaneously
- Cooperating threads depend on each other
- If one is missing, all others may slow down
- So, schedule them so they run all-or-none

Dedicated Processor Assignment

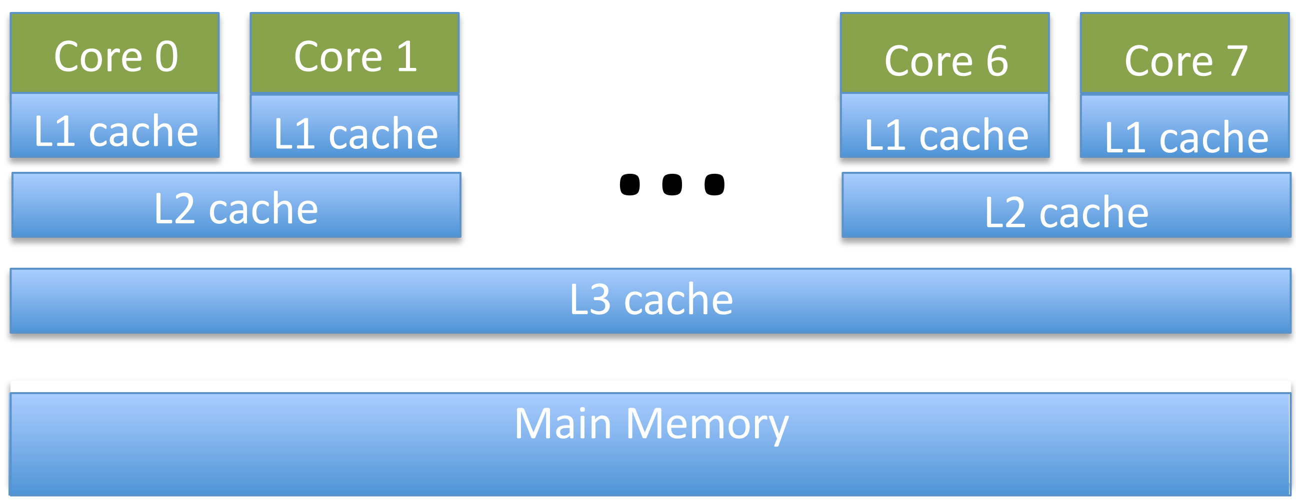
- Dedicate a group of CPUs to a multi-threaded process
- When process is scheduled, each thread is assigned to a CPU dedicated to that thread until process completes

Why Might This Be Good?

- If there are very large number of CPUs (e.g., hundreds), waste is not an issue
- No context switching
- A thread never waits for another *ready* thread (as all threads are “running”)

Cache Affinity

- Thread on same CPU benefits from cache
- In mul- core system
  – consider placement of threads on adjacent cores
  – however, must also consider cache contention



Real Time Scheduling

- Correctness of real-time systems depend on
  - logical result of computations
  - *and* the timing of these results
- Type of real-time systems
  - Hard vs. soft real-time
  - Periodic vs. aperiodic
- Scheduling
  - Earliest Deadline First (EDF)
  - Rate Monotonic Scheduling (RMS)

Real Time Scheduling

- Two types of latencies affect performance
  1. Interrupt latency – time from arrival of interrupt to start of routine that services interrupt
  2. Dispatch latency – time for schedule to take current process off CPU and switch to another

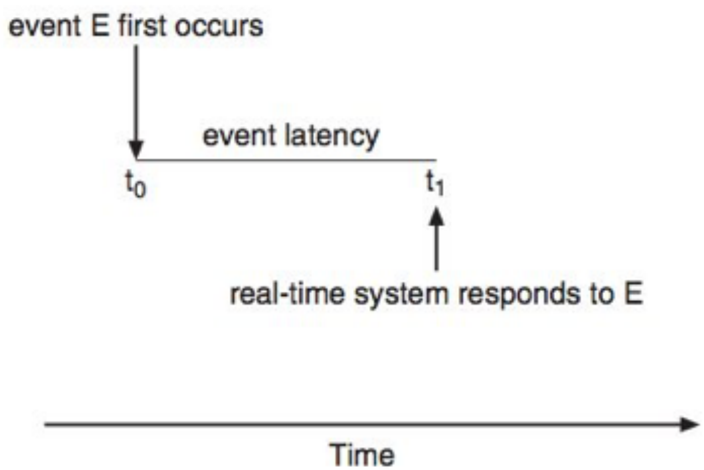


Figure 6.12 Event latency.

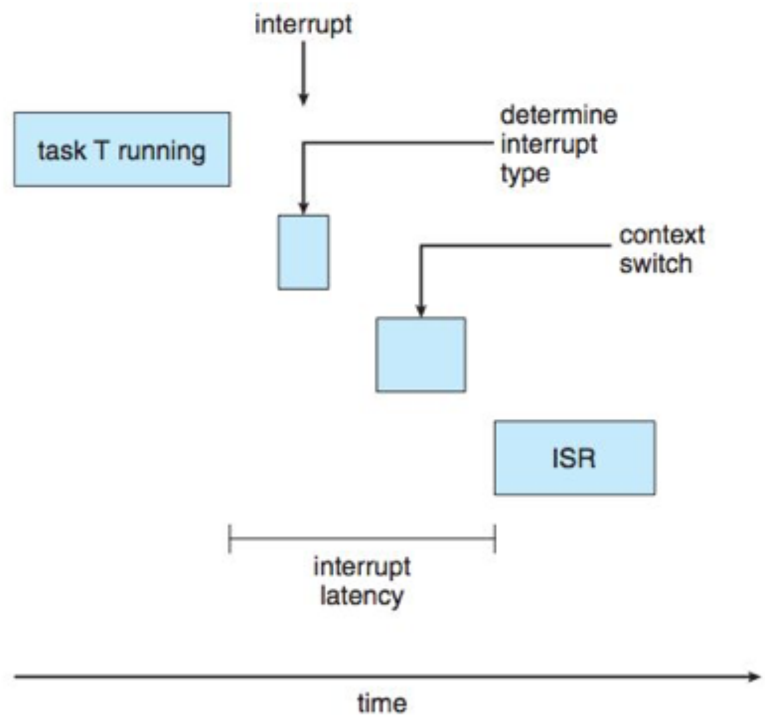
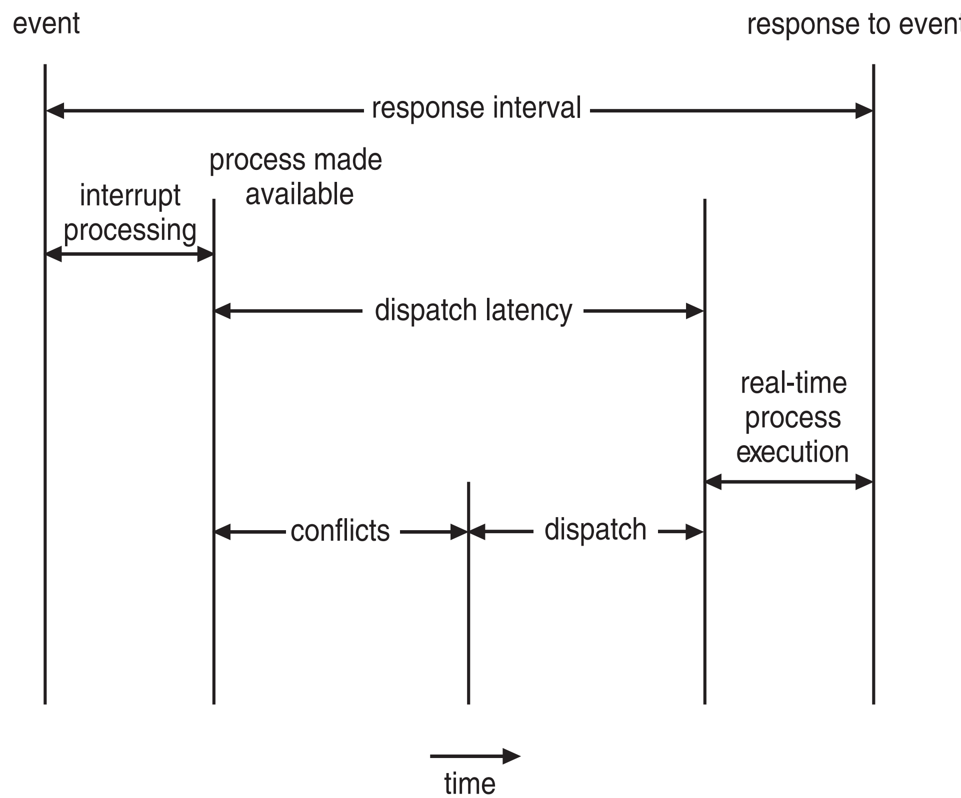


Figure 6.13 Interrupt latency.

Real Time Scheduling

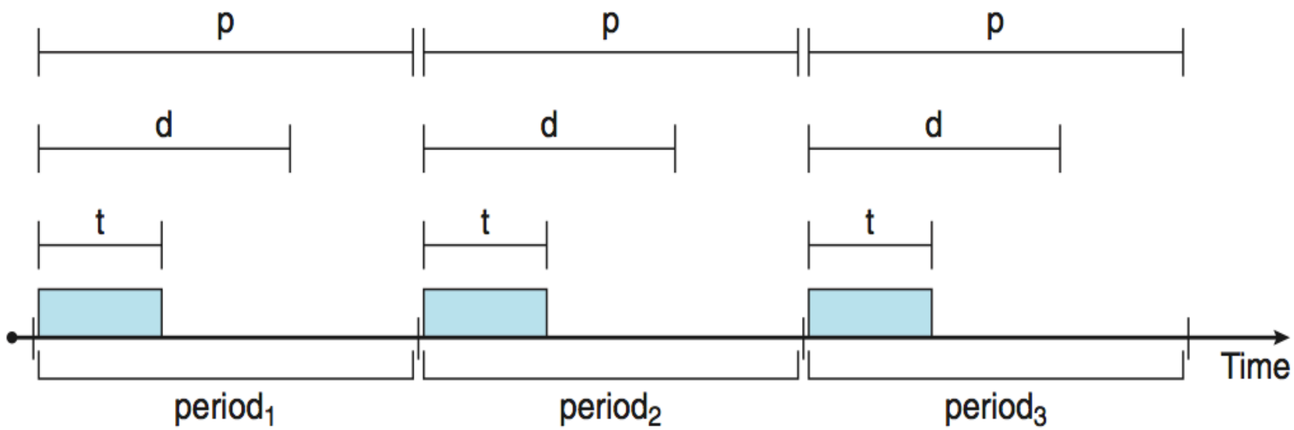
- Conflict phase of dispatch latency:

1. Preemption of any process running in kernel mode
2. Release by low-priority process of resources needed by high-priority processes

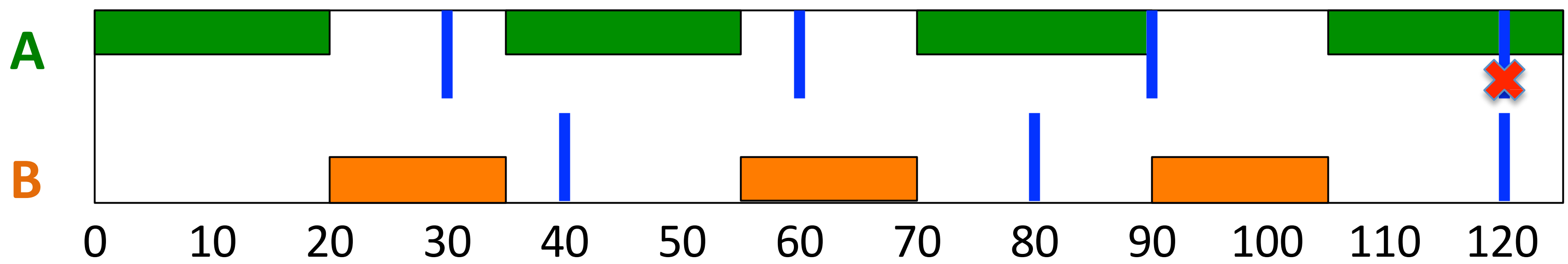


Priority-based Scheduling

- For real-time scheduling, scheduler must support preemptive, priority-based scheduling
    - But only guarantees soft real-time
- For hard real-time must also provide ability to meet deadlines
- Processes have new characteristics: **periodic** ones require CPU at constant intervals
    - Has processing time *t*, deadline *d*, period *p*
    - $0 \leq t \leq d \leq p$
    - **Rate** of periodic task is $1/p$



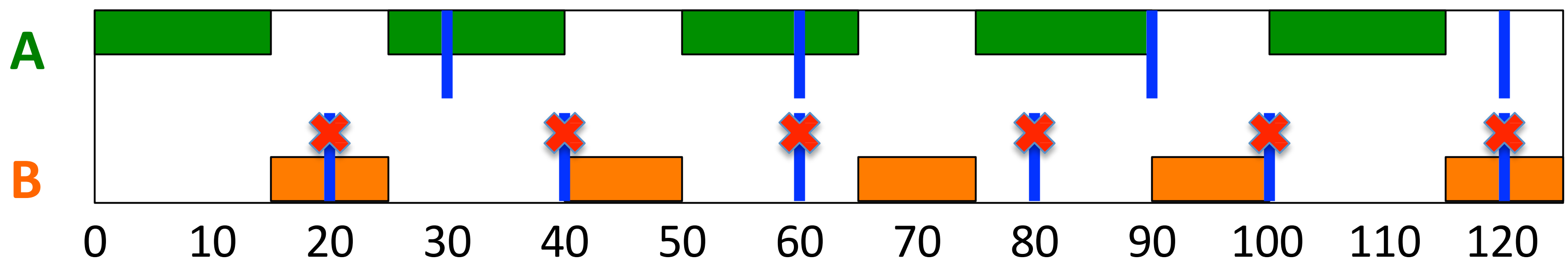
Periodic Threads (or Tasks)



- Periodic threads: computation is cyclic
- For each thread, given

  C = CPU burst, T = period, $U = C/T$ = utilization
- Can threads be ordered so deadlines are met?
- Consider orders: ABABAB…, vs. BABABA…

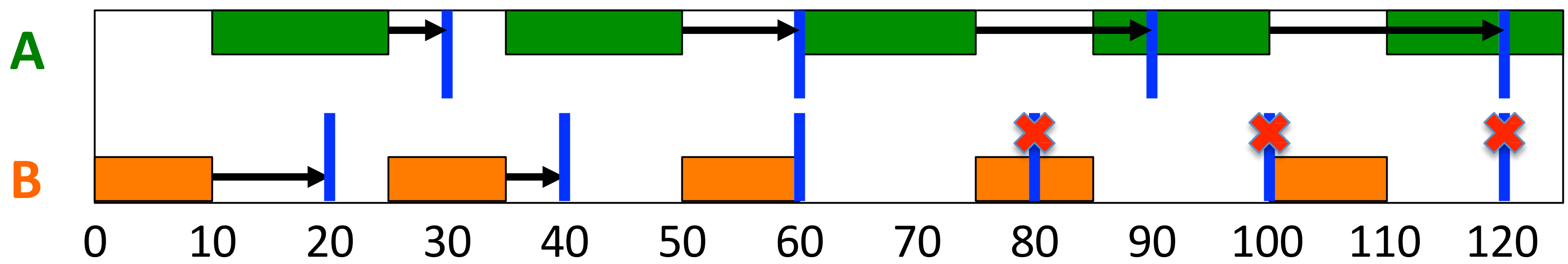
Periodic Threads



	C	*T*	*U*
A	15	30	50%
B	10	20	50%

- Sum of utilization does not exceed 100%
- For order ABABAB…, B misses all deadlines!

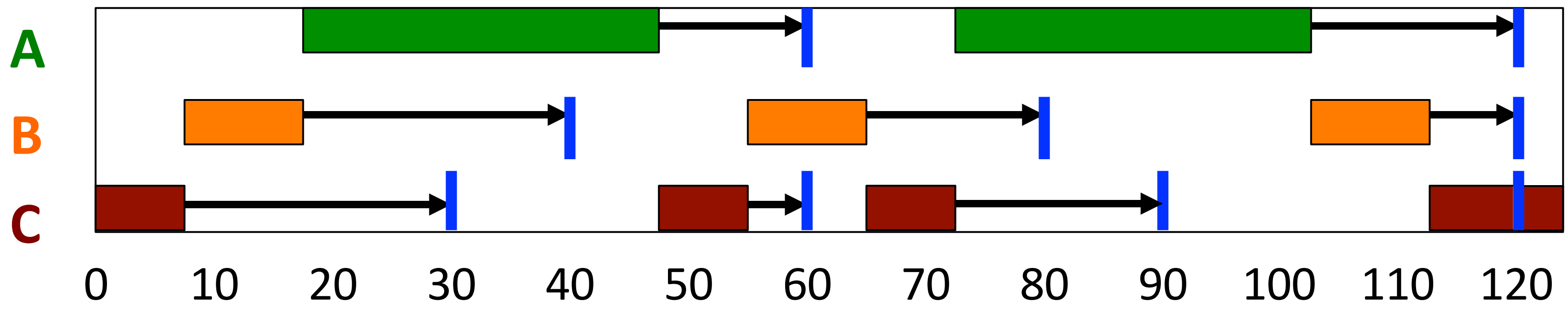
Periodic Threads



	C	*T*	*U*
A	15	30	50%
B	10	20	50%

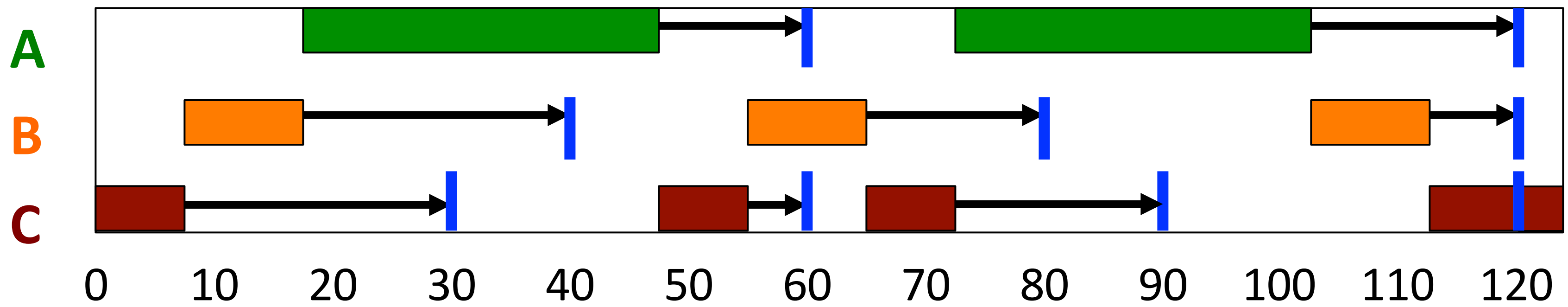
- Sum of utilizations does not exceed 100%
- For order BABABA…, B misses some deadlines

EDF: Earliest Deadline First



- Schedule thread with earliest deadline
- If earlier deadline thread appears, preempt
- Works for periodic and aperiodic threads
- Achieves 100% utilization (ignoring overhead!)
- Expensive: requires ordering by deadlines

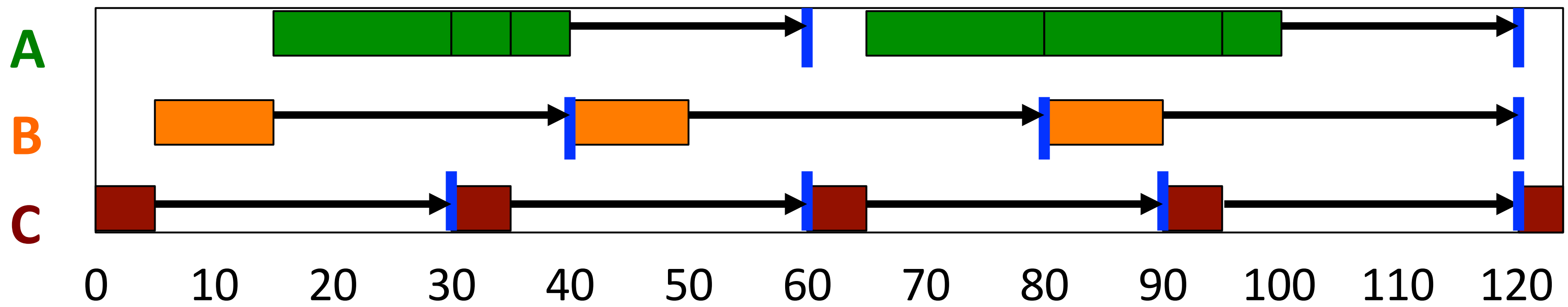
EDF: Earliest Deadline First



	C	*T*	*U*
A	30	60	50%
B	10	40	25%
C	7.5	30	25%

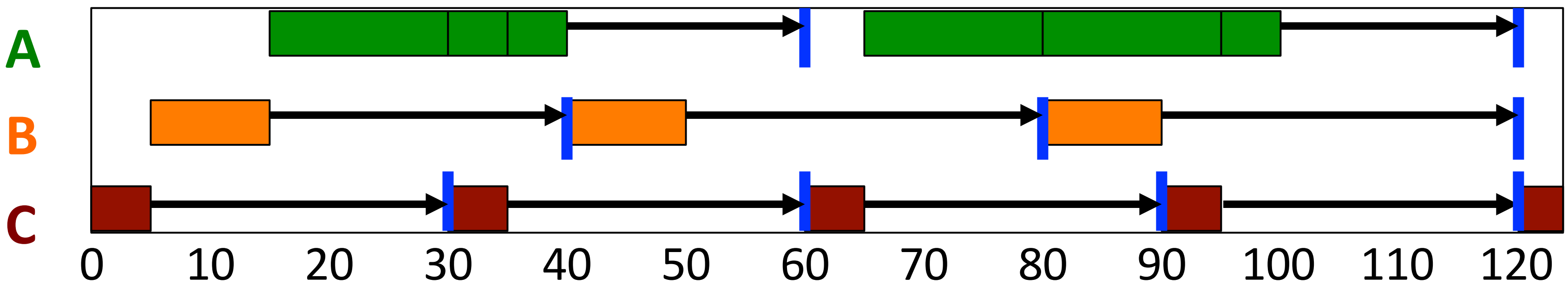
- Meets all deadlines, sum of utilizations = 100%

RMS: Rate Monotonic Scheduling



- If periodic threads, prioritize based on rates
- At start of period, select highest priority
- Preempt if necessary
- When burst done, wait till next period
- If $U_1 + \ldots + U_n \leq n\,(2^{1/n} - 1)$, all deadlines met

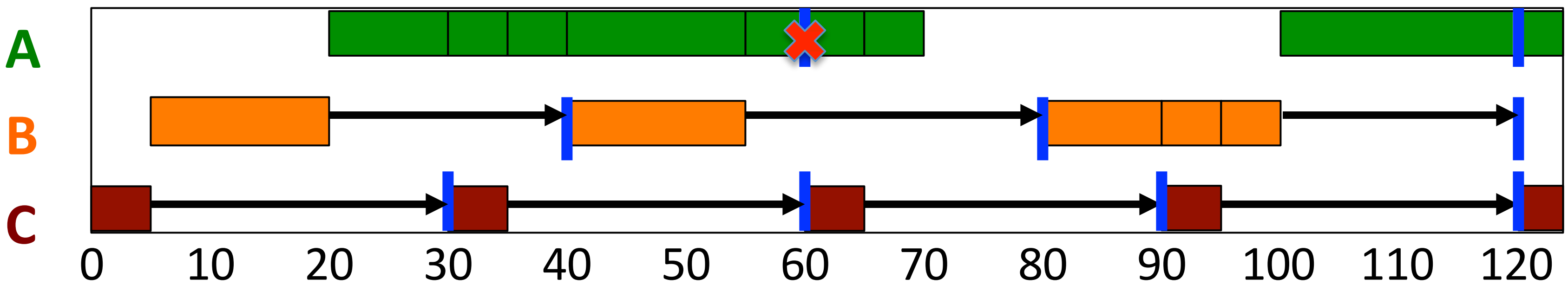
RMS Test Passes, All Deadlines Met



	C	*T*	*U*	Rate	Prio
A	20	60	33%	1/60 = 0.017	Low
B	10	40	25%	1/40 = 0.025	Med
C	5	30	17%	1/30 = 0.033	High

- Passes: $U_A + U_B + U_C = 75\% \leq 3\,(2^{1/3} - 1) = 78\%$

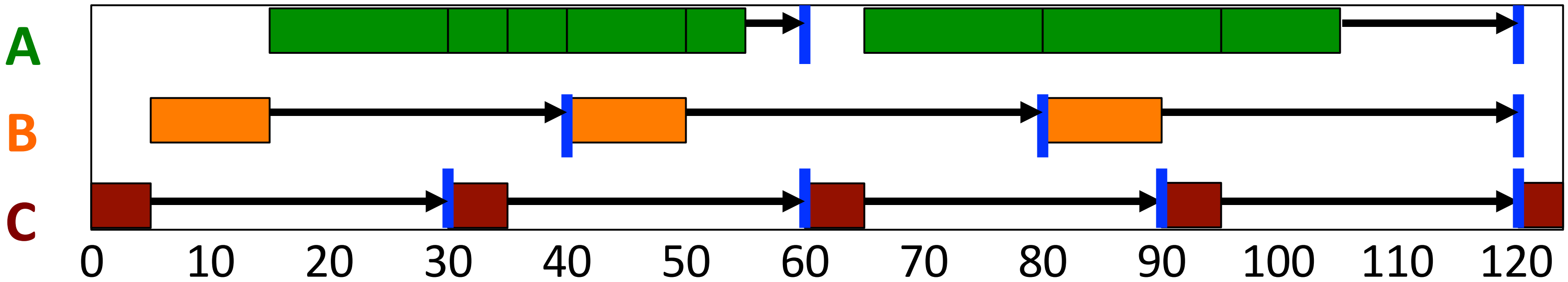
RMS Test Fails, Deadline Missed



	C	T	U	Rate	Prio
A	25	60	42%	1/60 = 0.017	Low
B	15	40	38%	1/40 = 0.025	Med
C	5	30	17%	1/30 = 0.033	High

- Fails: $U_A + U_B + U_C = 97\% > 3\ (2^{1/3} - 1) = 78\%$

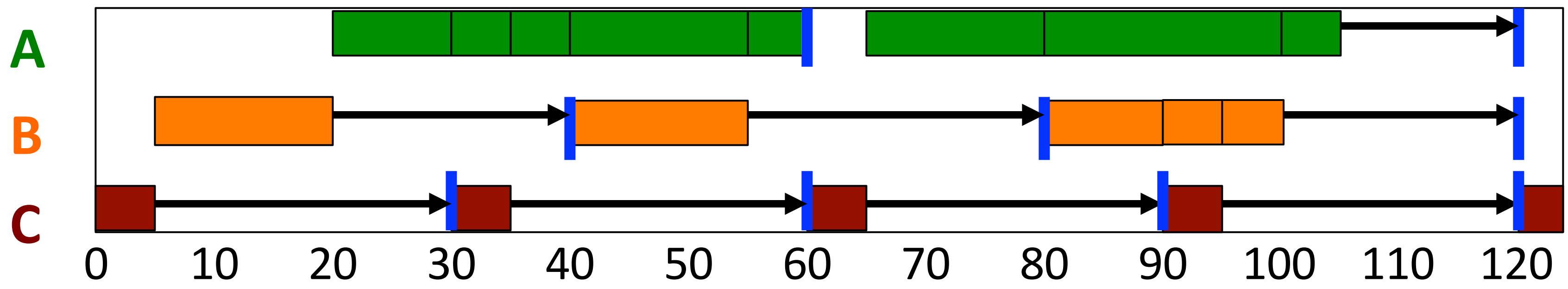
RMS Test Fails, But Deadlines Met



	C	*T*	*U*	Rate	Prio
A	25	60	42%	1/60 = 0.017	Low
B	10	40	25%	1/40 = 0.025	Med
C	5	30	17%	1/30 = 0.033	High

- Fails: $U_A + U_B + U_C = 84\% > 3\,(2^{1/3} - 1) = 78\%$

RMS Test Fails, But Deadlines Met



	C	*T*	*U*	Rate	Prio
A	20	60	33%	1/60 = 0.017	Low
B	15	40	38%	1/40 = 0.025	Med
C	5	30	17%	1/30 = 0.033	High

- Fails: $U_A + U_B + U_C = 88\% > 3\,(2^{1/3} - 1) = 78\%$

RMS Optimal But Limited

- RMS is simple and efficient
  - Static priority scheduling based on rates
- RMS is optimal for static priority algorithms
  - If RMS can't schedule, no other static priority can
- RMS is limited in what it guarantees
  - Utilization bounded by $n\,(2^{1/n} - 1) < \ln 2 \approx 69\%$
  - Deadlines will be met, but only if test passes
- RMS is limited to periodic threads

Summary

- Basic scheduling algorithms
- Shortest first/remaining time is best
- Mul- ple CPUs/Cores
  - Share queue
  - Peer approach
  - Co-schedule related threads
- Real– me scheduling: EDF and RMS