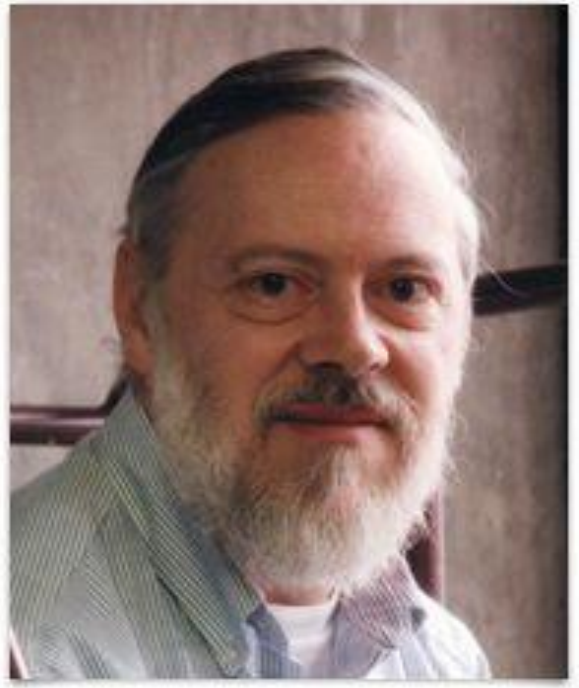
Embedded C Programming

REVIEW, DATA STRUCTURES

Brief History of C

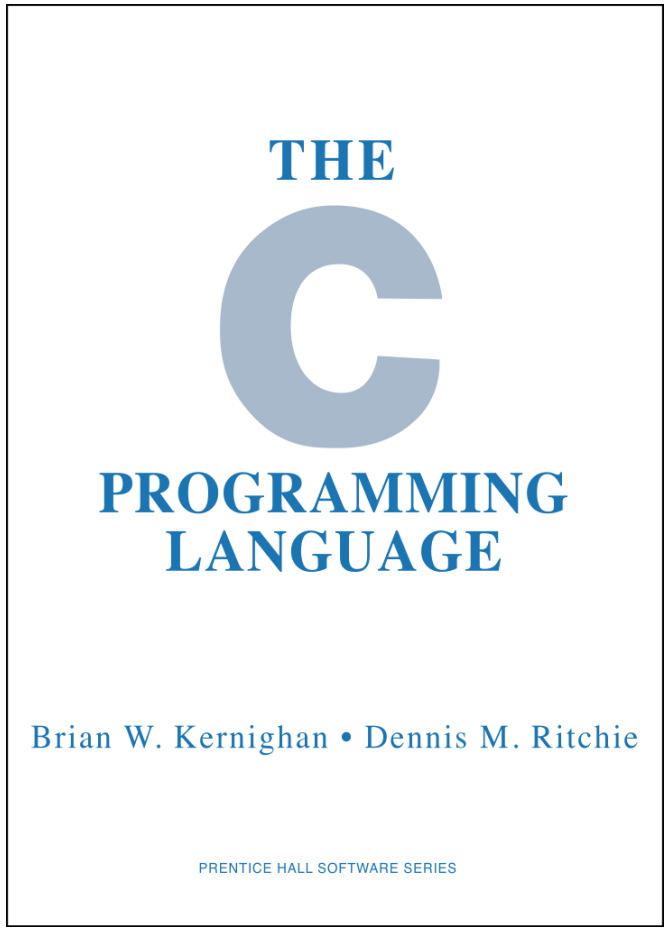
- Designed at AT&T Bell Laboratories by Dennis Ritchie in 1972
- Development closely related to UNIX
- Multiplatform
  - Language is simple to translate into assembly
  - Only 42 keywords in the core language



Dennis M. Ritchie
09/09/1941—10/12/2011

C Standards

- K&R C
  - Defacto standard for years
  - Described in the book "The C Programming Language" (K&R)
  - Not widely supported by modern compilers
- C89
  - ANSI C
  - Introduces the Standard C Library
  - Microsoft is stuck here with VS
- C99
  - Additions to the language
  - Deprecation of `gets()`



C Standards

- C11
  - Latest standard
  - Introduction of atomic operations
  - Threading support
  - Removal of `gets()`

```c
#include <stdio.h>
#include <stdlib.h>

int main(void) {
      fprintf(stdout, "hello, world\n");
      return EXIT_SUCCESS;
}
```

C and C++

- C++ is **not** an extension of the C language
- C++ is **not** C with classes
- C++ is its own language

```
/* this is valid C, but invalid C++ */
int* allocate_int_array(size_t n) {
      return malloc(n * sizeof(int));
}
/* this is valid C++, but invalid C */
int* allocate_int_array(size_t n) {
      return reinterpret_cast<int*>(malloc(n * sizeof(int)));
}
```

Working With C – Toolchain Workflow

- C Preprocessor runs in source file, handles directives

```
#include <inttypes.h>
#ifndef int32_t
#warning Needed type is not defined, falling back to least size.
#define int32_t int32_least_t
#endif
```

- C Compiler runs over the output of the C Preprocessor, converts C source into assembly listing
- Assembler runs over C Compiler output, generates object files
- Linker runs over the object files and creates a machine-dependent executable

Working With C – Behavior

- *Behavior*: external appearance or action
  - The action of integer addition has behavior which corresponds to arithmetic addition (with a few exceptions)
  - The operation of control-flow keywords
- *Undefined Behavior*: behavior, upon use of a nonportable or erroneous program construct or of erroneous data, for which there are **no** requirements
  - Dereferencing an invalid pointer
  - Overflow in signed integer arithmetic

Working With C – Behavior

- *Implementation Defined Behavior*: Specified behavior where each implementation documents how the choice is made
  - The placement of members on a bitfield
  - The addition of padding between types during allocation
- *Unspecified Behavior*: Use of an unspecified value, or other behavior where the standard provides two or more possibilities and imposes no further requirements on which is chosen in any instance
  - Order of evaluation of subexpressions in an expression

```
a = f() + g();  /* no specification on whether f() or g() is
                 * evaluated first */
```

Working With C – Behavior

- *Locale-specific behavior*: Behavior that depends on local conventions of nationality, culture, and language that each implementation documents
  - Using a comma as a decimal separator instead of a period

Working With C – Data Types

Data Type	bytes (bits)						
	Required	x86	x86_64	MSP430	AVR	ARM	SPARCv8
`short`	2 (16)	2 (16)	2 (16)	2 (16)	2 (16)	2 (16)	2 (16)
`int`	2 (16)	4 (32)	4 (32)	2 (16)	4 (32)	4 (32)	4 (32)
`long`	4 (32)	4 (32)	8 (64)	4 (32)	4 (32)	4 (32)	4 (32)
`long long`	8 (64)	8 (64)	8 (64)	8 (64)	8 (64)	8 (64)	8 (64)

- The `char` type is a special integer
  - Always 8 bits
  - Promoted to full integer under most operations
  - `char`, `signed char`, `unsigned char` are all different data types

Working With C – Data Types

- C99 **required** fixed width integer types
  - *Minimum width integers*
    - `int_least`N`_t`, `uint_least`N`_t`
    - Mandated for $N \in \{8,16,32,64\}$, other *N* optional
    - Underlying container may be wider than *N*
  - *Fastest type integer*
    - `int_fast`N`_t`, `uint_fast`N`_t`
    - Mandated for $N \in \{8,16,32,64\}$, other *N* optional
    - Underlying container may be wider than *N*
- C99 **optional** integers, not required to exist in the implementation
  - *Fixed width integers*
    - `int`N`_t`, `uint`N`_t`
    - Underlying container is exactly *N* bits wide

Working With C – Aggregate Types

- Combine primitive and other types into more complex types
  - structs
  - unions
  - bitfields
- Elements within an aggregate are called *members*
- Aggregates define necessary storage capacity
  - Compiler *must* know size to perform allocation

```
struct aggregate;
struct aggregate ag;
ag.member = 10; /* error: accessing member of incomplete type */
struct aggregate* agp = malloc(sizeof(*agp))  /* error: allocating
                                                 * incomplete type */
```

Working With C – Aggregate Types

- A `struct` is an aggregate whose storage capacity is *at least* that of the elements it contains
  - Compiler may add padding between members of the `struct` to preserve or add alignment
- A `union` is similar to a `struct`, however all members of a `union` occupy the same location in memory (overlap)
  - It is *undefined behavior* to assign to a member of a `union` and then read from another one

```
union u {
    int i;
    float f;
};
union u a;
a.f = 3.1415f;
printf("%d\n", a.i);  /* undefined behavior: assigned to a.f but
                       * reading from a.i */
```

Working with C – Aggregate Types

- A bitfield has no associated typename
  - Members of a union or `struct` may be declared to consist of a specified number of bits

```
struct packed_data {
    char a          : 4;
    char b          : 2;
    char c          : 2;
};
```

  - Compiler is free to change member order
  - Compiler is free to add padding between members

Working with C – Storage Classes

- `static`
  - Associated variable has permanent store
- `auto`
  - Associated variable has temporary store
  - Default for local variables
- `register`
  - Variable is to be register bound at all times
  - Implies `auto`
  - Can be ignored by modern compilers
- `_Thread_local`
  - Variable store is local to thread
  - Allocated when thread starts, deallocated when thread ends
- `extern`
  - Variable store is static or `_Thread_local` but no space is allocated for the variable in the current context
  - Backing store for the variable is provided by a different compilation unit

Working With C – Type Qualifiers

- `const`
  - Qualified type is read only
  - Attempts to modify an object defined with a `const`-qualified type through an `lvalue` with non-`const`-qualified type results in *undefined behavior*

```
const char* msg = "hello, world";
char* p = (char*)&msg[3];   /* discard const qualifier */
*p = 'L';                   /* undefined behavior */
```

Working With C – Type Qualifiers

- `volatile`
  - Accessing object with qualified type has side effects
  - What constitutes as a side effect is *implementation defined*
  - Attempts made to refer to an object defined with a `volatile`-qualified type through use of an `lvalue` with non-`volatile`-qualified type results in *undefined behavior*

```
volatile int i;
int* p = (int*)&i;      /* discard volatile qualifier */
*p = 10;                /* undefined behavior */

for(volatile int j = 0; j < 10; j++) {
    /* This loop has implementation defined behavior.
     * Different compilers may generate different outputs.
     * Do not use this as a delay loop. */
}
```

Working with C – Type Qualifiers

- `restrict`
  - Introduced in C99
  - Formal definition in ISO/IEC 9899:2011 Section 6.7.3.1
  - Pointers declared with this qualifier are guaranteed not to alias (point to same location in memory)
  - Guarantee is given by programmer
  - Hints a compiler to generate more efficient code
  - If pointers *do* alias, resulting code has *undefined behavior*
- `_Atomic`
  - Introduced in C11
  - Operations on objects whose type have been qualified as `_Atomic` are guaranteed to complete in a manner that appears instantaneous to the rest of the system

Working With C – Other Notes

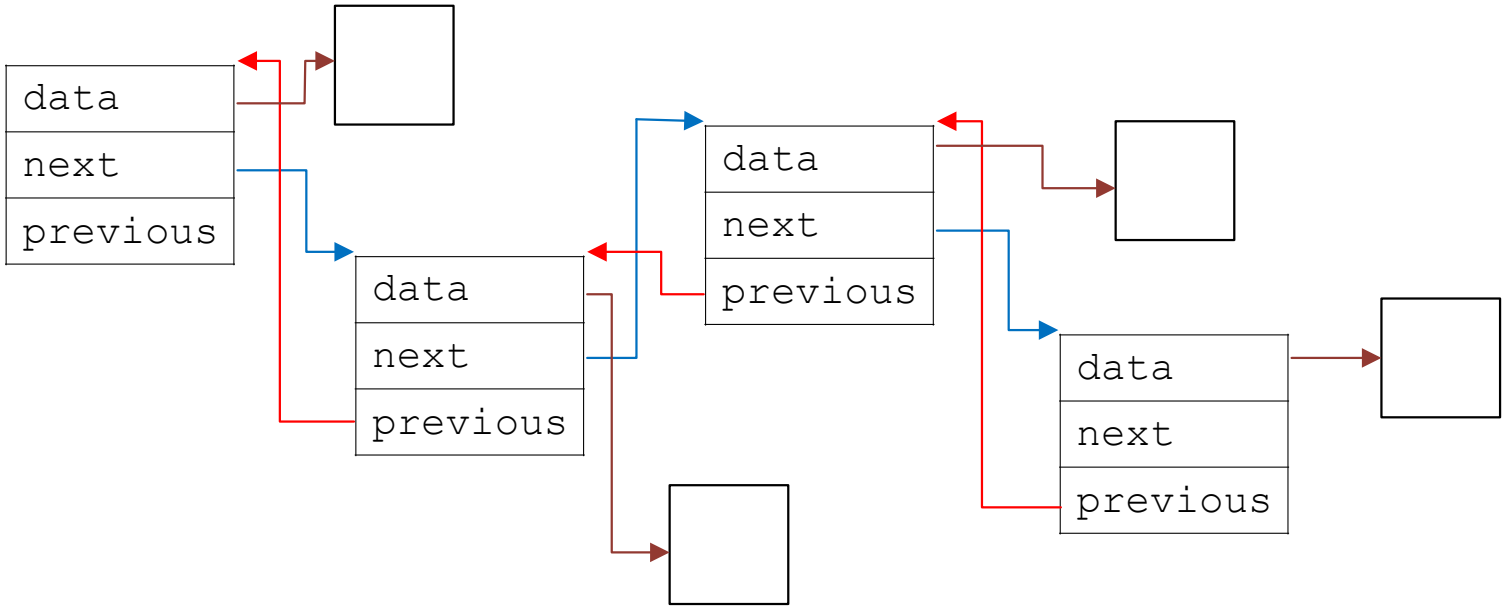
- Integer variables with permanent store that have not been explicitly initialized are automatically initialized to 0 by the runtime
  - In an embedded system, the linked-in runtime is responsible for this action
- Pointer variables with permanent store that have not been explicitly initialized are automatically initialized to NULL by the runtime
  - In an embedded system, the linked-in runtime is responsible for this action
- Local variables that have not been explicitly initialized have undefined contents

C Data Structures

- A linked list is a data structure that allows for theoretical infinite expansion
- It is often created as an aggregate, containing
  - a section for the data
  - a pointer to the next node on the list, and
  - optionally, a pointer to the previous node on the list
- Nodes on a linked list are often created dynamically

Linked Lists

```
struct double_linked_list_node {
      void* data;                              /* pointer to the data */
      struct linked_list_node* next;           /* next node */
      struct linked_list_node* previous;       /* previous node */
};
```



Linked Lists

- Traversal in $O(n)$
- Insertion in $O(1)$, provided insertion at head
- Deletion in $O(n)$, need to traverse list to find element

Linked Lists

```
/* allocate node with data, data must be previously allocated */
struct double_linked_list_node* allocate_node(void* data) {
	struct double_linked_list_node* t;
	t = (struct double_linked_list_node*)malloc(sizeof(*t));
	if(!t) {
		return (struct double_linked_list_node*)NULL;
	}
	t->next = (struct double_linked_list_node*)NULL;
	t->previous = (struct double_linked_list_node*)NULL;
	t->data = data;
	return t;
}
```

Linked Lists

```
/* insert data at start */
struct double_linked_list_node* insert_node(
            struct double_linked_list_node* head,
            void* data) {
      struct double_linked_list_node* t;
      if(!head) {
            return allocate_node(data);
      }
      t = allocate_node(data);
      /* should really check for allocation failure */
      t->next = head;
      head->previous = t;
      return t;
}
```

Linked Lists

```
/* search list for data */
struct double_linked_list_node* search_list(
            struct double_linked_list_node* head,
            const void* data) {

      struct double_linked_list_node* t;
      /* compare_data() returns 0 on match */
      for(t = head; t && compare_data(t->data, data); t = t->next) {
            ;
      }
      return t;
}
```

Linked Lists

```
/* delete first found element from list using data */
struct double_linked_list_node* delete_data(
            struct double_linked_list_node* head,
            const void* data) {

      struct double_linked_list_node* t;
      if(!head) {
            /* empty list */
            return (struct double_linked_list_node*)NULL;
      }
      if(!(t = search_list(head, data)) {
            /* no element to delete */
            return head;
      }
      /* continues… */
```

Linked Lists

```
        if(t->next) {
                /* if not at the tail, bridge over next element */
                t->next->prev = t->prev;
        }
        if(t->prev) {
                /* if we are not the head of the list, tie the previous
                 * node to the next
                 */
                t->prev->next = t->next;
        } else {
                /* we are the head of the list, new head needed */
                head = t->next;
        }
        free(t);
        /* return the possibly new head */
        return head;
}
```

Real-Time Systems and Scheduling

THEORY, & ALGORITHMS

Real-Time Systems

- Correctness of the system depends on
  - *Logical results* of computation
  - *Time* at which the results are produced
- Tasks need to complete before a *deadline*
  - System is at *fault* otherwise
  - Task not completing before deadline is a *scheduling failure*
- To guarantee the timing behavior, system must be *predictable*
  - When a task is activated, it should be possible to determine its completion time with certainty
  - Upper bound suffices for most cases

Real-Time Systems

A system is real-time *if for a set of tasks* $T = \{\tau_1, \tau_2, \tau_3, \dots, \tau_n\}$ *where the worst case execution time for task* $\tau_i \in T$ *is* C_i *if there exists at least one task* τ_c *with a deadline* D_c *so that one of the following conditions is met:*

- τ_c *is completed before its deadline, that is* $C_c \leq D_c$*. In this case, the system is called* hard real-time*.*
- τ_c *is completed sometime after its deadline, that is* $C_c \leq D_c$*. A penalty,* $P(t_c)$*, is paid if the scheduling constrain is not met. In this case, the system is called* soft real-time*.*
- τ_c *is completed before its deadline, that is* $C_c \leq D_c$*. A reward function* $R(\tau_c)$ *is defined so that if the task is completed after its deadline it becomes 0. Otherwise the reward is a positive function.*

Tasks in Real-Time Systems

Tasks in real-time systems can be:

- *Periodic*
  - Become ready regularly at a fixed rate
  - Usually constrained to execute within one period *P*
  - Deadline is *P*
- *Aperiodic*
  - Activate irregularly at some unknown, possibly unbound rate
  - Constrained by deadline *D*
- *Sporadic*
  - Activated irregularly, with some known bound rate
  - Characterized by a minimum interval between activations
  - Constrained by a deadline *D*

Tasks in Real-Time Systems

Tasks can also be

- *Preempted* if they can be interrupted when a task with equal or higher priority becomes ready
  - So far, our scheduler preempts all (user) tasks
- *non-preemptive* if they should be completed without interruption

Task priority can be

- *Static* if they are assigned and never changed
- *Dynamic* if priority can change as the system runs

Tasks in Real-Time Systems

Tasks can also be

- *Independent* if they can be executed without regards to other tasks
- *Dependent* if executing a task is dependent on a certain resource or condition becoming available

The Job of the Scheduler

The act of deciding which runnable task is to be executed is called *scheduling*. Formally,

Given a set of tasks $T = \{\tau_1, \tau_2, \dots, \tau_n\}$, *a set of processors* $\pi = \{\pi_1, \pi_2, \dots, \pi_m\}$, *and a set of resources* $R = \{R_1, R_2, \dots, R_k\}$, scheduling *refers to the act of assigning tasks from* T *to processors from* π *and resources from* R *so that all tasks complete under certain imposed constraints.*

Schedulers

- Non-real-time
  - Round Robin scheduler
- Real-time
  - Rate Monotonic scheduler
  - Deadline Monotonic scheduler

Round Robin Scheduling

Given a set of tasks $T = \{\tau_1, \tau_2, \ldots \tau_n\}$, each task τ_i is given equal CPU time without regards for priority.

- Start at τ_1 and allow it to run for P, then
- Switch to τ_2 and allow it to run for P, then
- …
- Switch to τ_n and allow it to run for P, then
- Switch to τ_1 and allow it to run for P, then…

Round Robin Scheduling

Task	Period	Arrival	Burst Length
τ_1	80	0	30
τ_2	40	8	10
τ_3	100	0	15

One CPU Quantum:
5 ticks

0	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30	32	34	36	38
40	42	44	46	48	50	52	54	56	58	60	62	64	66	68	70	72	74	76	78
80	82	84	86	88	90	92	94	96	98	100	102	104	106	108	110	112	114	116	118

Round Robin Scheduling

Task	Period	Arrival	Burst Length
τ_1	80	0	30
τ_2	40	8	10
τ_3	100	0	15

One CPU Quantum:
5 ticks

Run Queue at T = 0: $\{\tau_1, \tau_3\}$

0	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30	32	34	36	38

τ_1, τ_3 arrive, scheduler picks τ_1 (τ_3 is a valid option)

40	42	44	46	48	50	52	54	56	58	60	62	64	66	68	70	72	74	76	78

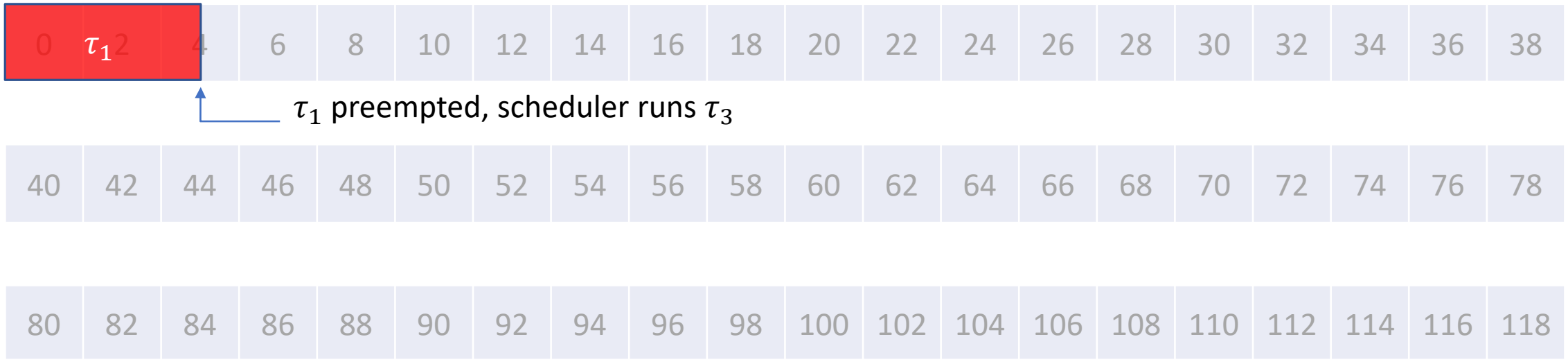
80	82	84	86	88	90	92	94	96	98	100	102	104	106	108	110	112	114	116	118

Round Robin Scheduling

Task	Period	Arrival	Burst Length
τ_1	80	0	30
τ_2	40	8	10
τ_3	100	0	15

One CPU Quantum:
5 ticks

Run Queue at T = 5: $\{\tau_1, \tau_3\}$

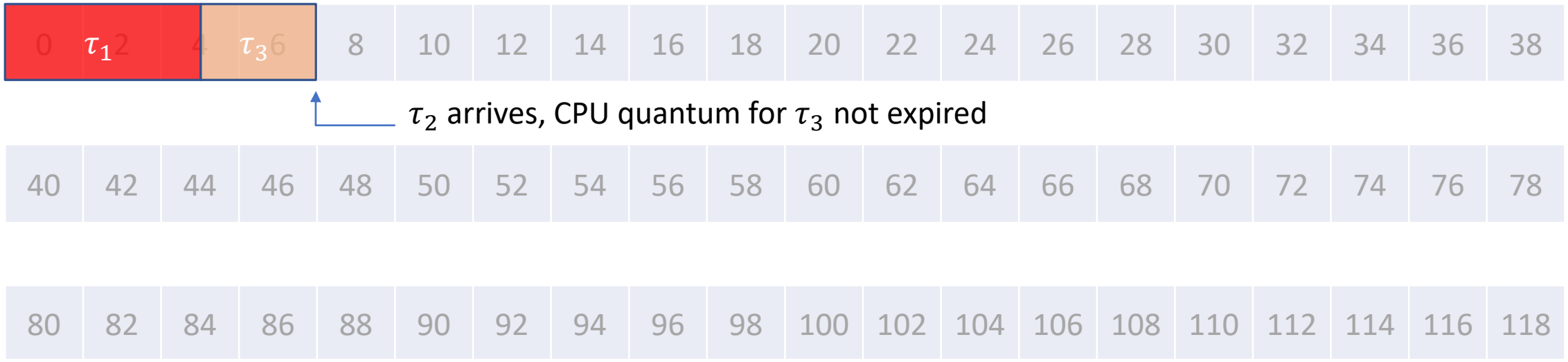


Round Robin Scheduling

Task	Period	Arrival	Burst Length
τ_1	80	0	30
τ_2	40	8	10
τ_3	100	0	15

One CPU Quantum:
5 ticks

Run Queue at T = 8: $\{\tau_1, \tau_3, \tau_2\}$

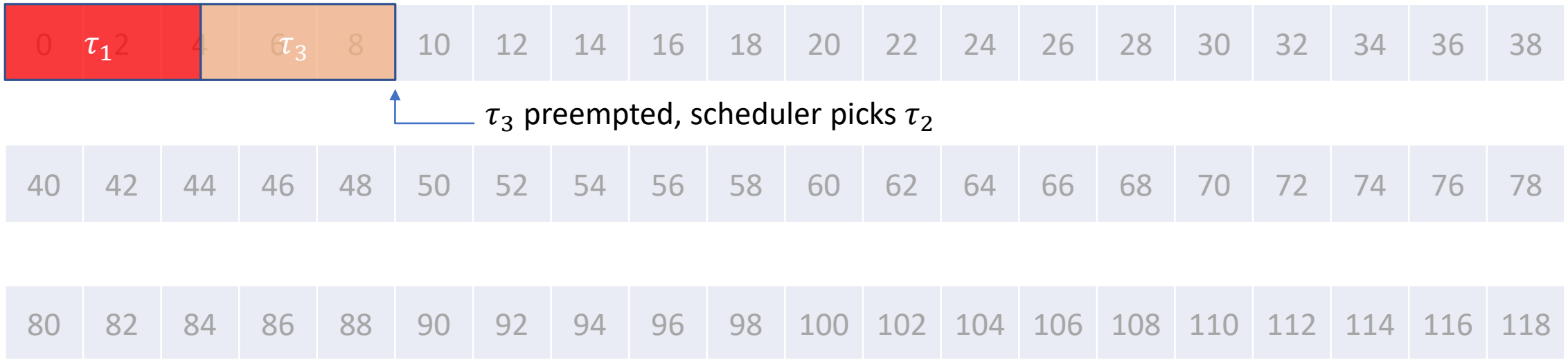


Round Robin Scheduling

Task	Period	Arrival	Burst Length
τ_1	80	0	30
τ_2	40	8	10
τ_3	100	0	15

One CPU Quantum:
5 ticks

Run Queue at T = 10: $\{\tau_1, \tau_3, \tau_2\}$

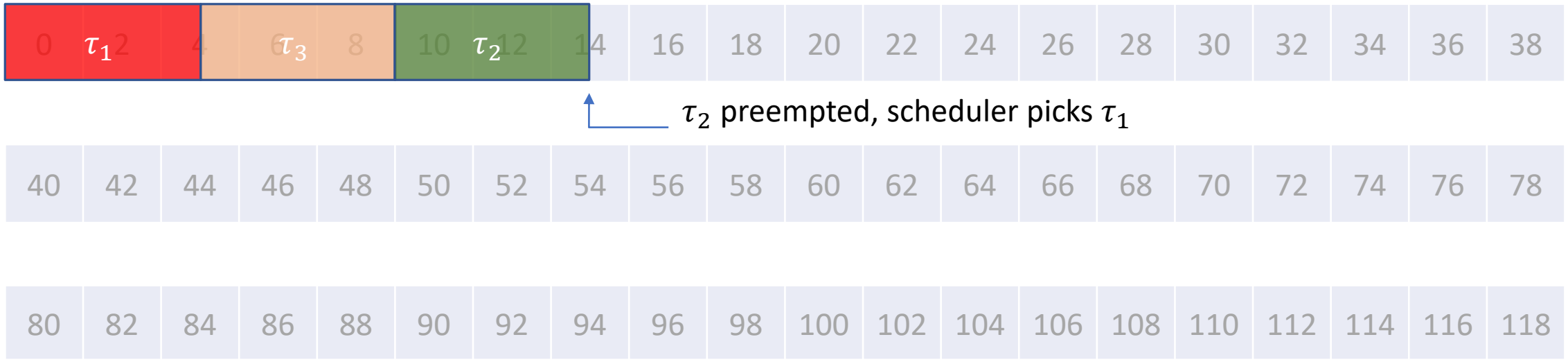


Round Robin Scheduling

Task	Period	Arrival	Burst Length
τ_1	80	0	30
τ_2	40	8	10
τ_3	100	0	15

One CPU Quantum:
5 ticks

Run Queue at T = 15: $\{\tau_1, \tau_3, \tau_2\}$

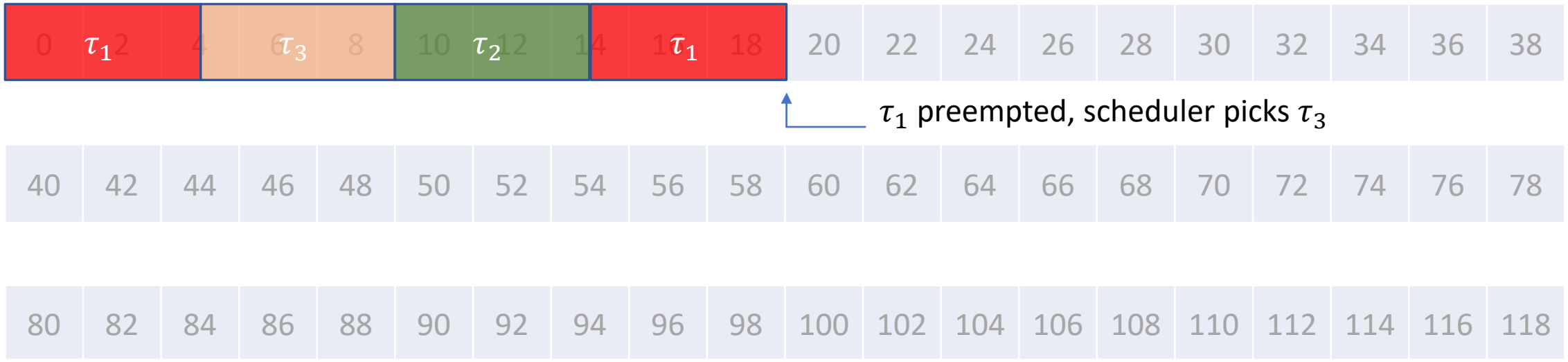


Round Robin Scheduling

Task	Period	Arrival	Burst Length
τ_1	80	0	30
τ_2	40	8	10
τ_3	100	0	15

One CPU Quantum:
5 ticks

Run Queue at T = 20: $\{\tau_1, \tau_3, \tau_2\}$



Round Robin Scheduling

Task	Period	Arrival	Burst Length
τ_1	80	0	30
τ_2	40	8	10
τ_3	100	0	15

One CPU Quantum:
5 ticks

Run Queue at T = 25: $\{\tau_1, \tau_3, \tau_2\}$

40 42 44 46 48 50 52 54 56 58 60 62 64 66 68 70 72 74 76 78

80 82 84 86 88 90 92 94 96 98 100 102 104 106 108 110 112 114 116 118

Round Robin Scheduling

Task	Period	Arrival	Burst Length
τ_1	80	0	30
τ_2	40	8	10
τ_3	100	0	15

One CPU Quantum:
5 ticks

Run Queue at T = 30: $\{\tau_1, \tau_3\}$

40 42 44 46 48 50 52 54 56 58 60 62 64 66 68 70 72 74 76 78

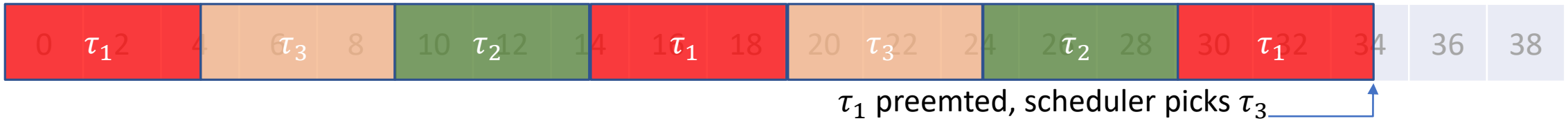
80 82 84 86 88 90 92 94 96 98 100 102 104 106 108 110 112 114 116 118

Round Robin Scheduling

Task	Period	Arrival	Burst Length
τ_1	80	0	30
τ_2	40	8	10
τ_3	100	0	15

One CPU Quantum:
5 ticks

Run Queue at T = 35: $\{\tau_1, \tau_3\}$



40 42 44 46 48 50 52 54 56 58 60 62 64 66 68 70 72 74 76 78

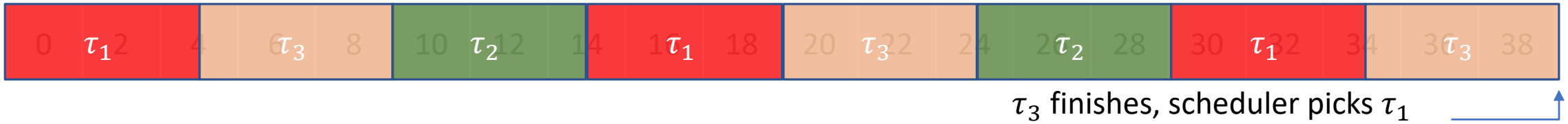
80 82 84 86 88 90 92 94 96 98 100 102 104 106 108 110 112 114 116 118

Round Robin Scheduling

Task	Period	Arrival	Burst Length
τ_1	80	0	30
τ_2	40	8	10
τ_3	100	0	15

One CPU Quantum:
5 ticks

Run Queue at T = 40: $\{\tau_1\}$



40 42 44 46 48 50 52 54 56 58 60 62 64 66 68 70 72 74 76 78

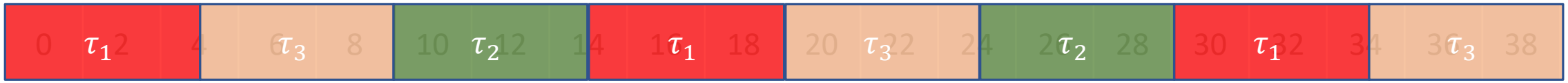
80 82 84 86 88 90 92 94 96 98 100 102 104 106 108 110 112 114 116 118

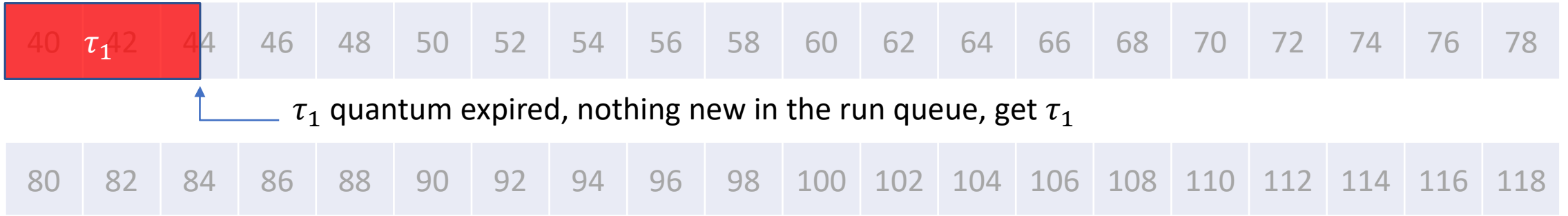
Round Robin Scheduling

Task	Period	Arrival	Burst Length
τ_1	80	0	30
τ_2	40	8	10
τ_3	100	0	15

One CPU Quantum:
5 ticks

Run Queue at T = 45: $\{\tau_1\}$



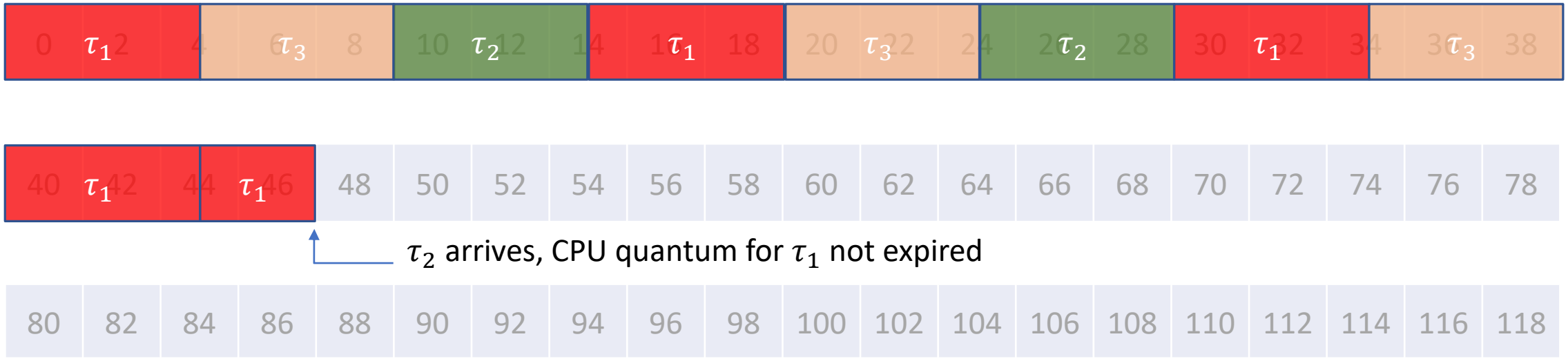


Round Robin Scheduling

Task	Period	Arrival	Burst Length
τ_1	80	0	30
τ_2	40	8	10
τ_3	100	0	15

One CPU Quantum:
5 ticks

Run Queue at T = 48: $\{\tau_1, \tau_2\}$

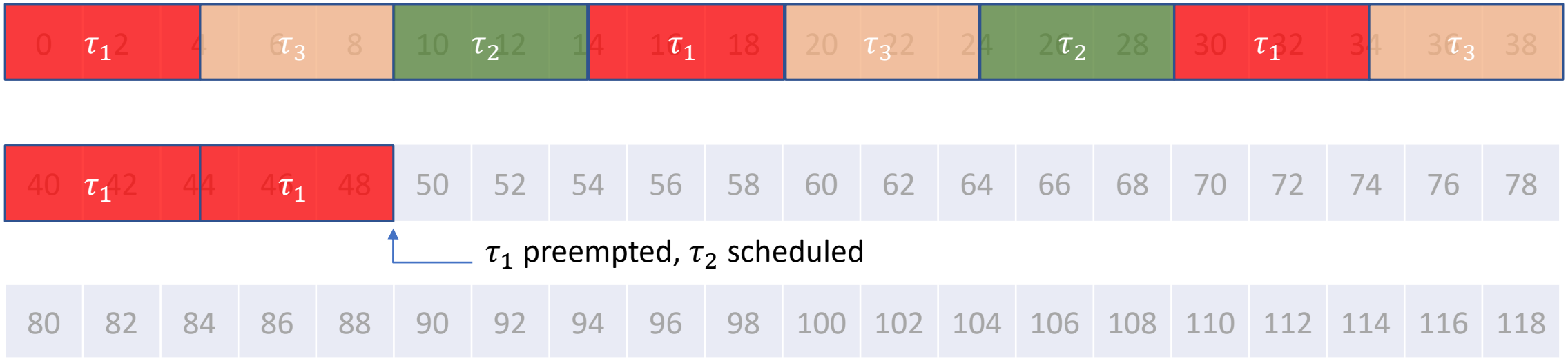


Round Robin Scheduling

Task	Period	Arrival	Burst Length
τ_1	80	0	30
τ_2	40	8	10
τ_3	100	0	15

One CPU Quantum:
5 ticks

Run Queue at T = 50: $\{\tau_1, \tau_2\}$

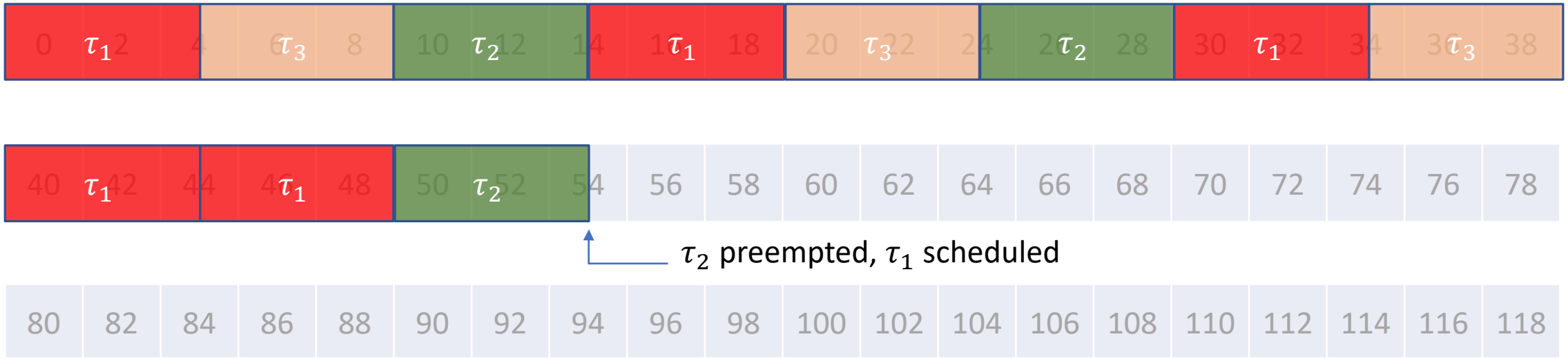


Round Robin Scheduling

Task	Period	Arrival	Burst Length
τ_1	80	0	30
τ_2	40	8	10
τ_3	100	0	15

One CPU Quantum:
5 ticks

Run Queue at T = 55: $\{\tau_1, \tau_2\}$



Round Robin Scheduling

Task	Period	Arrival	Burst Length
τ_1	80	0	30
τ_2	40	8	10
τ_3	100	0	15

One CPU Quantum:
5 ticks

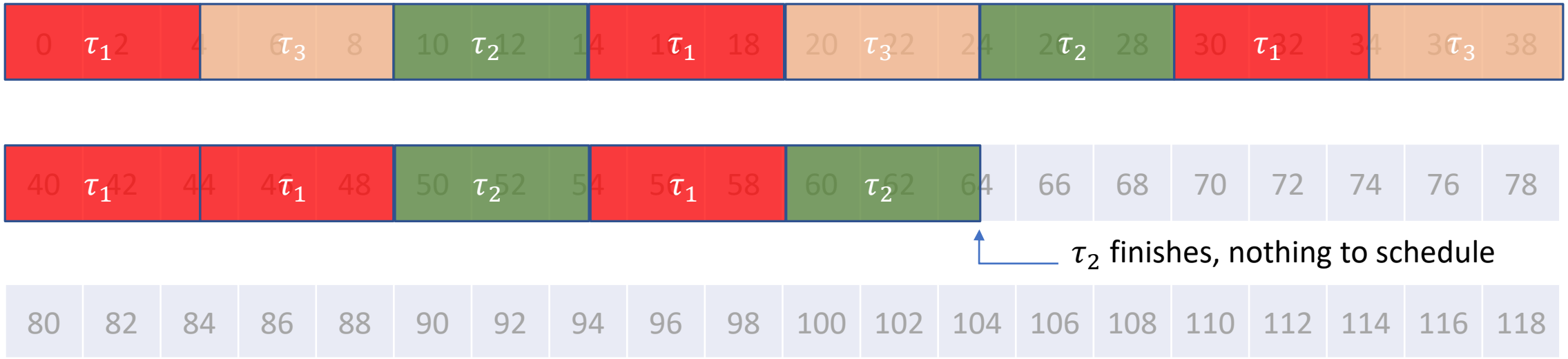
Run Queue at T = 60: $\{\tau_2\}$

Round Robin Scheduling

Task	Period	Arrival	Burst Length
τ_1	80	0	30
τ_2	40	8	10
τ_3	100	0	15

One CPU Quantum:
5 ticks

Run Queue at T = 65: $\emptyset$

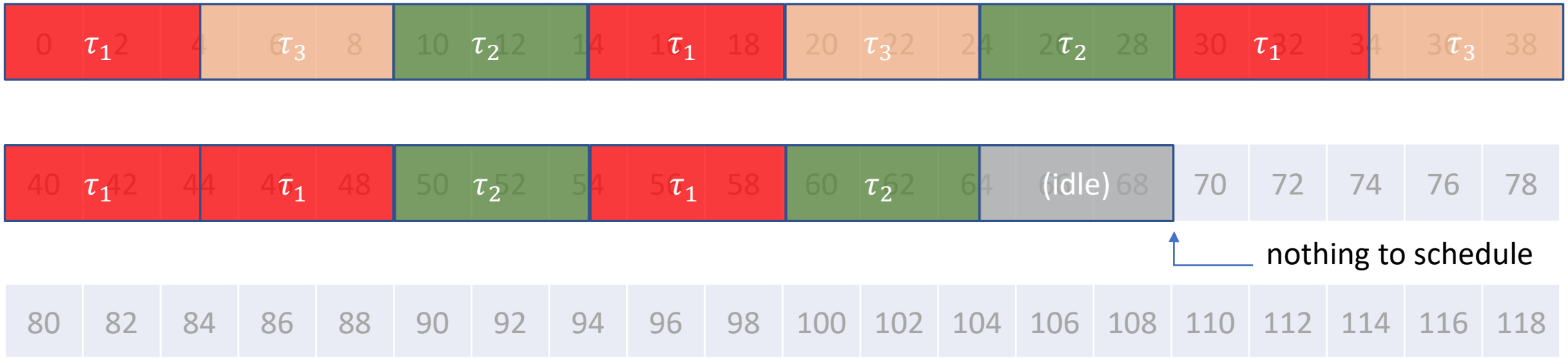


Round Robin Scheduling

Task	Period	Arrival	Burst Length
τ_1	80	0	30
τ_2	40	8	10
τ_3	100	0	15

One CPU Quantum:
5 ticks

Run Queue at T = 70: ∅

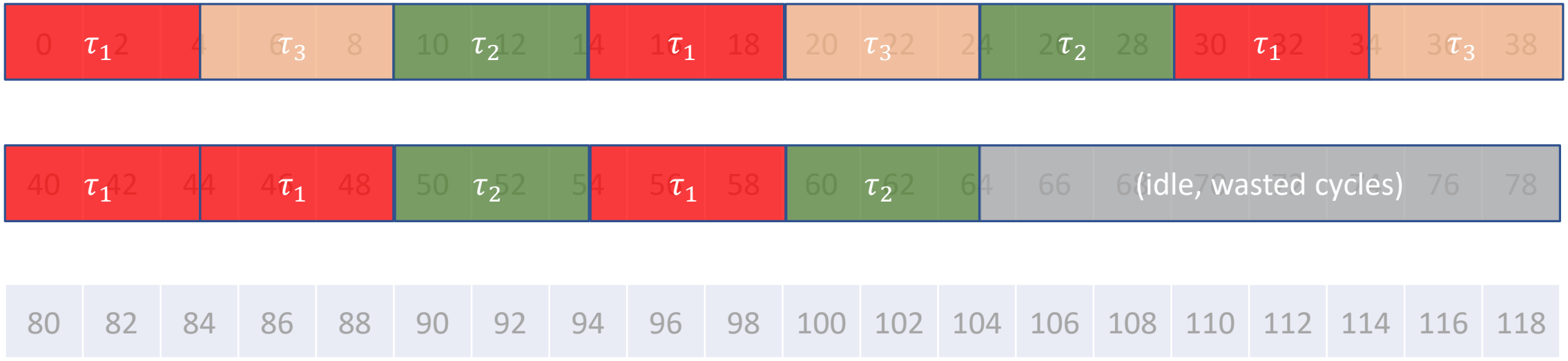


Round Robin Scheduling

Task	Period	Arrival	Burst Length
τ_1	80	0	30
τ_2	40	8	10
τ_3	100	0	15

One CPU Quantum:
5 ticks

Run Queue at T = 80: $\{\tau_1\}$

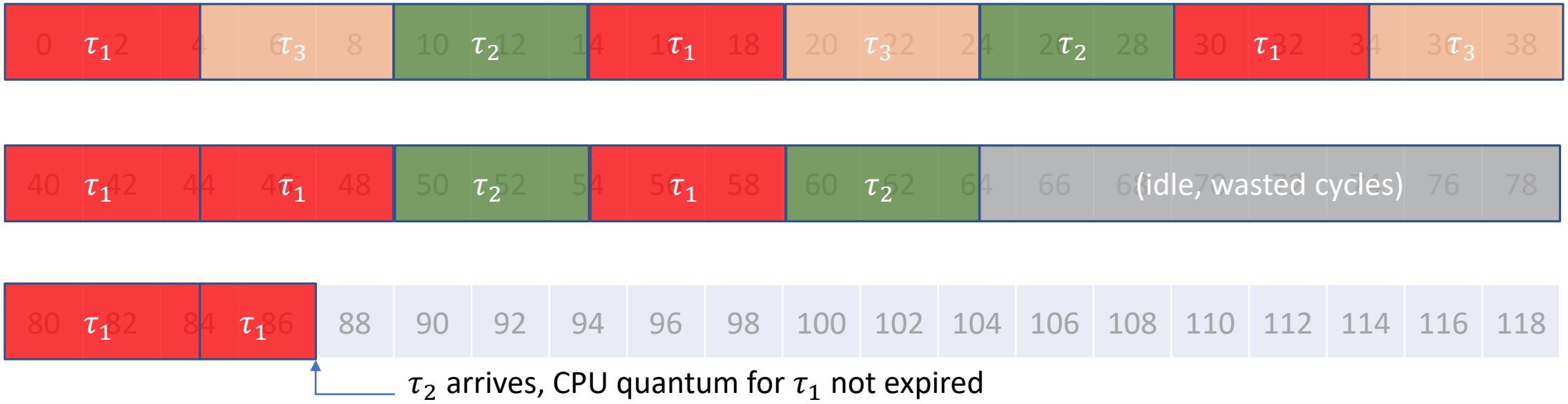


Round Robin Scheduling

Task	Period	Arrival	Burst Length
τ_1	80	0	30
τ_2	40	8	10
τ_3	100	0	15

One CPU Quantum:
5 ticks

Run Queue at T = 88: $\{\tau_1, \tau_2\}$

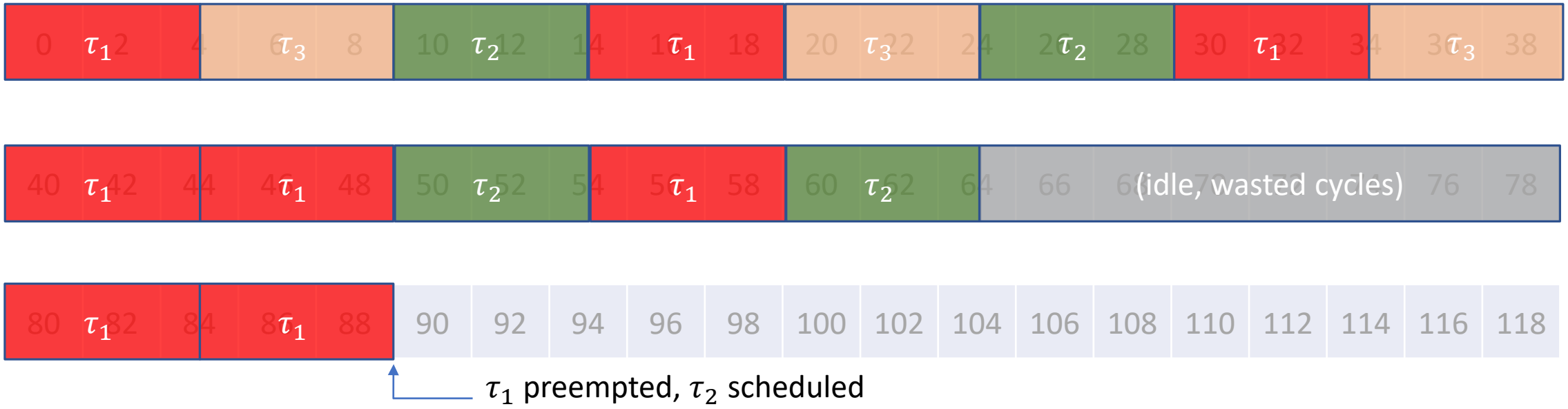


Round Robin Scheduling

Task	Period	Arrival	Burst Length
τ_1	80	0	30
τ_2	40	8	10
τ_3	100	0	15

One CPU Quantum:
5 ticks

Run Queue at T = 90: $\{\tau_1, \tau_2\}$

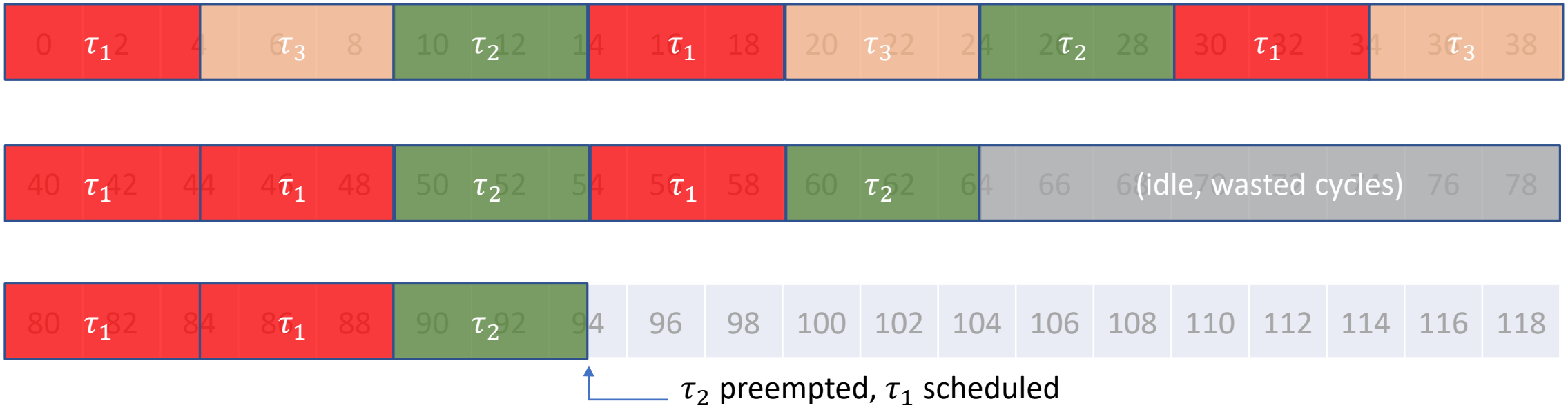


Round Robin Scheduling

Task	Period	Arrival	Burst Length
τ_1	80	0	30
τ_2	40	8	10
τ_3	100	0	15

One CPU Quantum:
5 ticks

Run Queue at T = 95: $\{\tau_1, \tau_2\}$

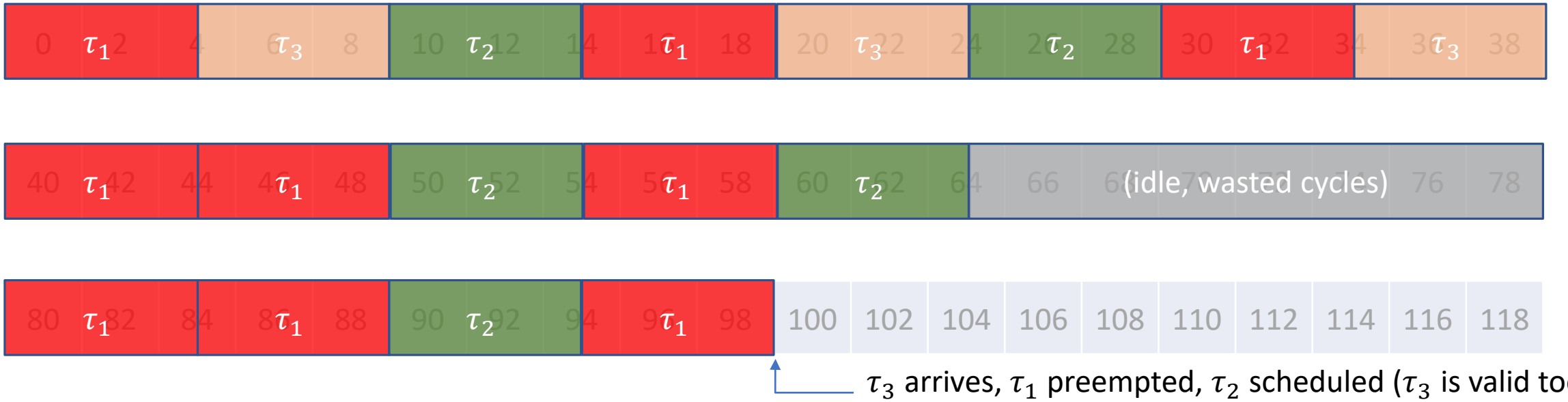


Round Robin Scheduling

Task	Period	Arrival	Burst Length
τ_1	80	0	30
τ_2	40	8	10
τ_3	100	0	15

One CPU Quantum:
5 ticks

Run Queue at T = 100: $\{\tau_1, \tau_2, \tau_3\}$



CPU Efficiency

- Idle cycles are wasted cycles, the CPU could be doing something at this point, but it is doing nothing
- Measure CPU usage

$$U = \sum_{i=1}^{n} \frac{C_i}{P_i}$$

  - C_i is the execution time of the task (burst length) τ_i
  - P_i is the period of task τ_i

CPU Usage

Task	Period	Arrival	Burst Length
τ_1	80	0	30
τ_2	40	8	10
τ_3	100	0	15

$$U = \frac{30}{80} + \frac{10}{40} + \frac{15}{100} = 77.5\%$$

CPU Usage

Task	Period	Arrival	Burst Length
τ_1	80	0	30
τ_2	40	8	10
τ_3	100	0	15

$$U = \frac{30}{80} + \frac{10}{40} + \frac{15}{100} = 77.5\%$$

CPU is idle 22.5% of the time. We want CPU utilization to be as close to 100% as possible.

Round Robin Scheduling

Given:

circular list of tasks *task_list*

size of task list *N*

number of scheduling ticks *t*

schedule:

current ← task_list[t]

t ← (t + 1) mod N

Context Switching

- *Context Switching* refers to the act of storing the active execution context (CPU state) and restoring a new context
- Scheduler swaps tasks to time-share the CPU
- Scheduler must context switch between tasks
  - Save all current task's registers
  - Save current apsr
  - Restore next task's apsr
  - Restore next task's registers
- Use PendSV to initiate the process

Context Switching

- Recall that on entering handler mode CPU saves `lr`, `pc`, `xPSR`, `r0 – r3`, `r12`
- Software responsible for saving `r4—r11`
- Before exiting handler mode
  - Software restores new `r4—r11`
  - CPU restores new `lr`, `pc`, `xPSR`, `r0—r3`, `r12`

Use PendSV to call schedule!

```
        .globl pendsv_vect
        .type pendsv_vect, %function
        .align 4
pendsv_vect:
        push {lr}
        mrs r0, psp
        stmdb r0!, {r4-r11}
        msr psp, r0
        bl schedule
        ldmia r0!, {r4-r11}
        msr psp, r0
        pop {pc}
```

Use SysTick to request PendSV

```
/* 100 SysTick interrupts per CPU quantum */
#define TICKS_PER_QUANTUM 100

#define ICSR (*((volatile unsigned int*)(0xe000ed04)))
#define ICSR_PENDSVSET (1 << 28)

static volatile unsigned int ticks_count;

void systick_handler(void) {
      ticks_count++;
      if(ticks_count == TICKS_PER_QUANTUM) {
            /* time for the scheduler to run! */
            ticks_count = 0;
            ICSR |= ICSR_PENDSVSET;
      }
      /* handle other SysTick related events */
}
```

Basic Semaphores

SEMAPHORES, SPIN-LOCKS

Semaphores

- Count the number of resources available on system
- Block access to resource if the resource has been exhausted
  - System may have maximum number of files that can be opened at a time
  - Attempt to open a new file may block until a handler becomes available for use

Semaphores

- Semaphores are equipped with two operations
  - V: increments (*verhogen*)
  - P: decrement (Dijkstra: *prolaag*, short for *prober te verlagen*, try to reduce)
- Semaphore operations must be atomic in nature
- The value of the semaphore is the number of units that are available
  - P operation wastes time until a resource becomes available

Semaphores

operation V(semaphore S):

S ← S + 1

operation P(semaphore S):

atomic repeat:

if S ≥ 1 **then** { S ← S – 1; **break** }

Semaphores

- Producer-consumer problem
  - One process (producer) generates data
  - One process (consumer) receives data
  - Communication is achieved using a queue of size N
- Rules of communication
  - Consumer must wait for the producer if queue is empty
  - Producer must wait for consumer if queue is full

Semaphores

- Track the state of the queue using two semaphores
  - `empty_count`: number of empty places in queue
  - `full_count`: number of elements in queue
- Use binary semaphore (mutex) to ensure queue integrity
  - `use_queue`

Semaphores

producer:

```
P(empty_count)
P(use_queue)
add_to_queue(item)
V(use_queue)
V(full_count)
```

consumer:

```
P(full_count)
P(use_queue)
item ← get_from_queue()
V(use_queue)
V(empty_count)
```

- `empty_count` is initialized to the number of slots in the queue
- `full_count` is initially 0
- `use_queue` is initially 1

Trivial Solution

- Ensure that producer/consumer can not be interrupted in critical section
  - Disable interrupts at start of critical section
  - Enable them at end of critical section

Trivial Solution

```
/* int start_critical_section(void) */
start_critical_section:
        mrs r0, primask
        cpsid i
        bx lr
```

```
/* void end_critical_section(int) */
end_critical_section:
        msr primask, r0
        bx lr
```

```
/* program code */
int t;
t = start_critical_section();
/* critical section */
end_critical_section(t);
```

Issues

- We are disabling interrupts
  - Can miss interrupt timings and thus miss events
- The `msr` and `mrs` instructions are privileged
  - Can only run if the CPU is in privileged mode
  - In non-privileged mode, a fault will be generated
  - Can try moving stuff to a service call, but we disable interrupts… How do we get back into the kernel after we exit it with interrupts disabled?
- This only works in *uniprocessor systems*
  - In a multiprocessor system, we only disable interrupts in one core
  - Scheduled task in second core can run its critical section
  - Can try disabling interrupts in all cores, but then we need to ensure mutual exclusion of tasks in the scheduler

Semaphores in Thumb2

- Implementing semaphores:
  - `ldrex{cond} rt, [rn {,#offset}]`
    - load register exclusive
    - if the physical address has the Shared TLB attribute, the instruction tags the physical address as excusive access for the current processor and clears other exclusive access tag for any other physical address
    - otherwise, it tags that the executing processor has an outstanding tagged physical address
  - `strex{cond} rd, rt, [rn {, #offset}]`
    - store register exclusive
    - performs a conditional store to memory
      - if the physical address does not have the Shared TLB attribute, and the executing processor has an outstanding tagged physical address, the store takes place, the tag is cleared, and the value 0 is returned in `rd`
      - if the physical address does not have the Shared TLB attribute, and the executing processor does not have an outstanding physical address, the store does not take place, and the value 1 is returned in `rd`
      - if the physical address has the Shared TLB attribute, and the physical address is tagged as exclusive access for the executing processor, the store takes place, the tag is cleared, and the value 0 is returned in `rd`
      - if the physical address has the shared TLB attribute, and the physical address is not tagged as exclusive access for the executing processor, the store does not take place and the value 1 is returned in `rd`

Semaphores in Thumb2

```
/* spinlock wait loop*/
        ldr r2, =lock_address
        mov r1, #1
1:      ldrex r0, [r2]              /* get value of lock, place tag on it */
        cmp r0, #0                  /* check if zero */
        bne 1b                      /* if not zero, someone else has lock */
        strex r0, r1, [r2]          /* try to store lock on it, if we lost
                                     * the tag because someone else read
                                     * from it, the store will fail
                                     */
        cmp r0, #0                  /* check if the store succeeded */
        bne 1b                      /* if not, we try again */
        /* we now have the lock, access critical resource */
```

More on Scheduling

REAL-TIME SCHEDULERS, PERIODIC EVENTS

Rate Monotonic Scheduling

- Real time priority scheduler
- The task with the shortest period is scheduled first
- Task is run until it finishes
- Running task is preempted by one with higher priority

Rate Monotonic Scheduling

Task	Period	Arrival	Burst Length
τ_1	80	0	25
τ_2	50	0	10
τ_3	100	0	15

0	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30	32	34	36	38
40	42	44	46	48	50	52	54	56	58	60	62	64	66	68	70	72	74	76	78
80	82	84	86	88	90	92	94	96	98	100	102	104	106	108	110	112	114	116	118

Rate Monotonic Scheduling

Task	Period	Arrival	Burst Length
τ_1	80	0	25
τ_2	50	0	10
τ_3	100	0	15

Run Queue at T = 0: $\{\tau_1, \tau_2, \tau_3\}$

0	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30	32	34	36	38

τ_1, τ_2, τ_3 arrive, τ_2 has shortest period thus is scheduled

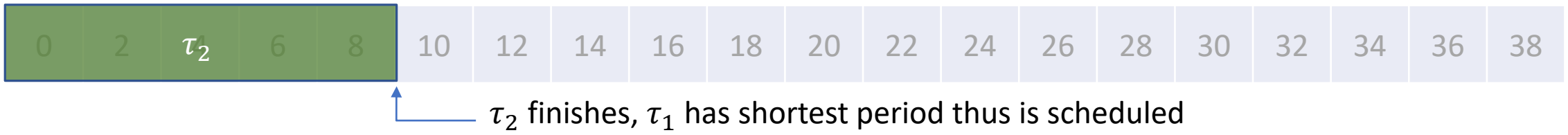
40	42	44	46	48	50	52	54	56	58	60	62	64	66	68	70	72	74	76	78

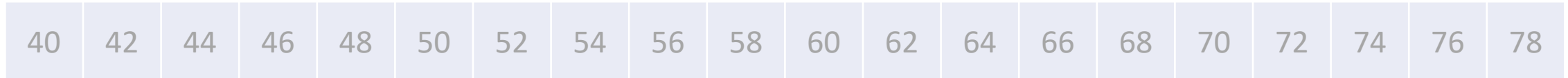
80	82	84	86	88	90	92	94	96	98	100	102	104	106	108	110	112	114	116	118

Rate Monotonic Scheduling

Task	Period	Arrival	Burst Length
τ_1	80	0	25
τ_2	50	0	10
τ_3	100	0	15

Run Queue at T = 10: $\{\tau_1, \tau_3\}$

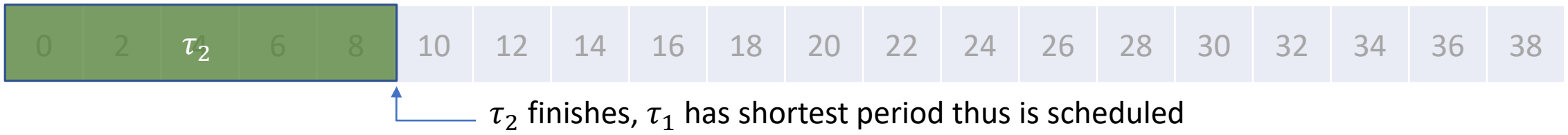


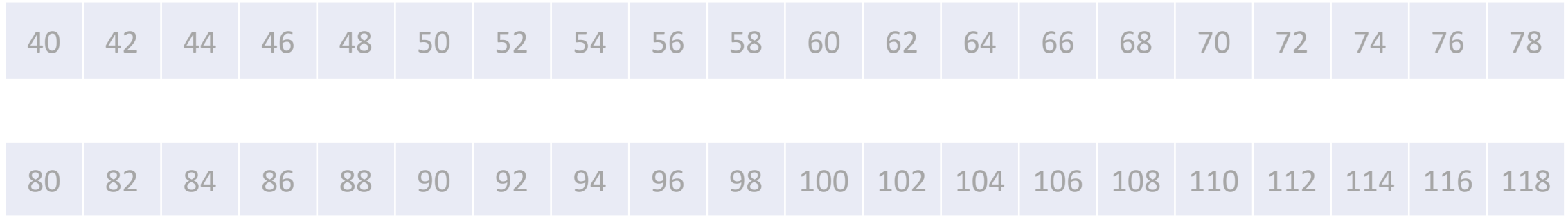


Rate Monotonic Scheduling

Task	Period	Arrival	Burst Length
τ_1	80	0	25
τ_2	50	0	10
τ_3	100	0	15

Run Queue at T = 10: $\{\tau_1, \tau_3\}$

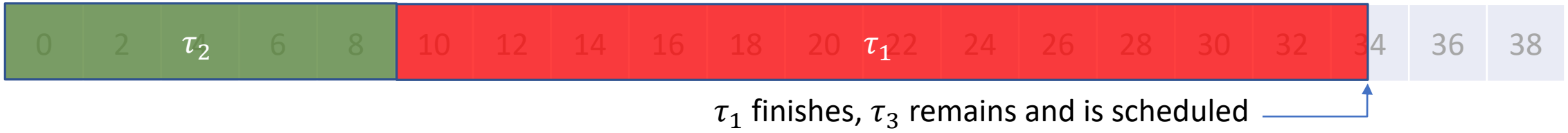




Rate Monotonic Scheduling

Task	Period	Arrival	Burst Length
τ_1	80	0	25
τ_2	50	0	10
τ_3	100	0	15

Run Queue at T = 35: $\{\tau_3\}$



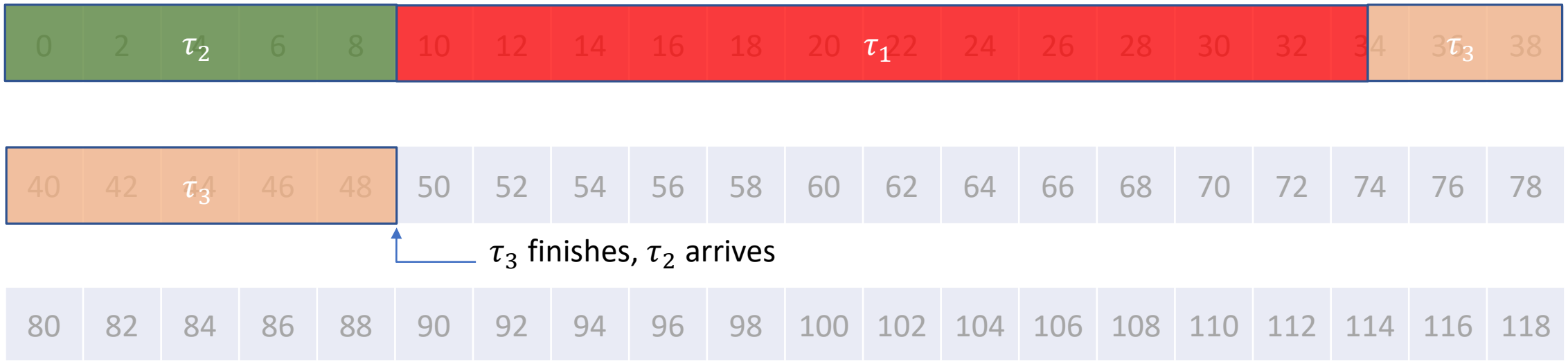
40 42 44 46 48 50 52 54 56 58 60 62 64 66 68 70 72 74 76 78

80 82 84 86 88 90 92 94 96 98 100 102 104 106 108 110 112 114 116 118

Rate Monotonic Scheduling

Task	Period	Arrival	Burst Length
τ_1	80	0	25
τ_2	50	0	10
τ_3	100	0	15

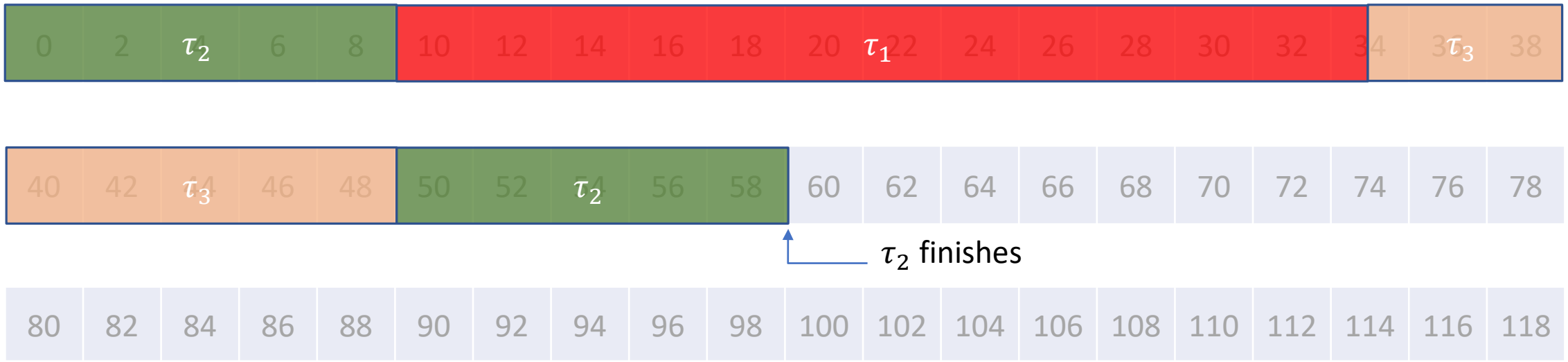
Run Queue at T = 35: $\{\tau_2\}$



Rate Monotonic Scheduling

Task	Period	Arrival	Burst Length
τ_1	80	0	25
τ_2	50	0	10
τ_3	100	0	15

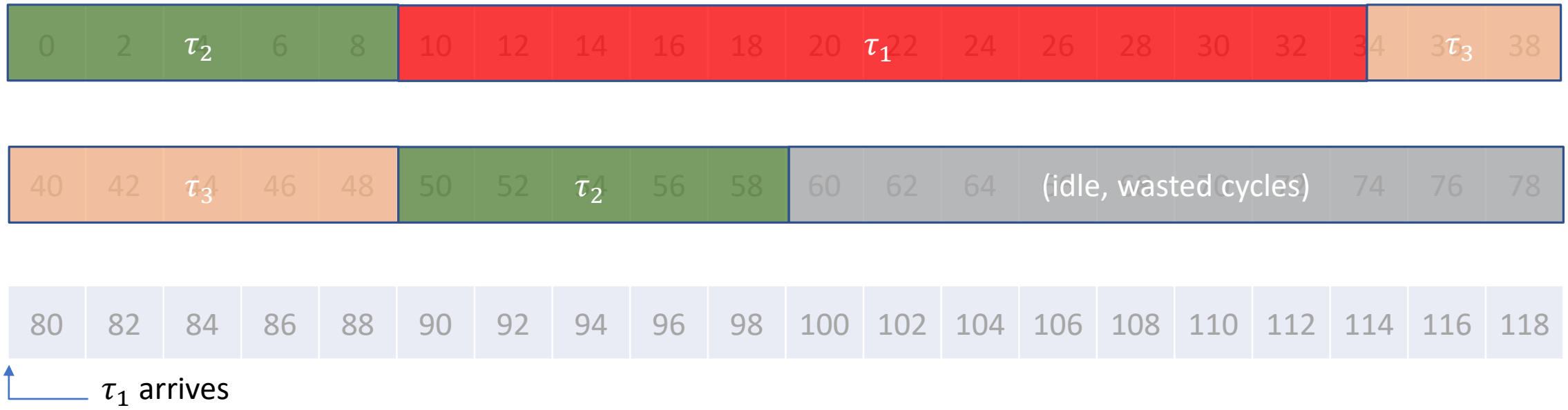
Run Queue at T = 60: $\emptyset$



Round Robin Scheduling

Task	Period	Arrival	Burst Length
τ_1	80	0	25
τ_2	50	0	10
τ_3	100	0	15

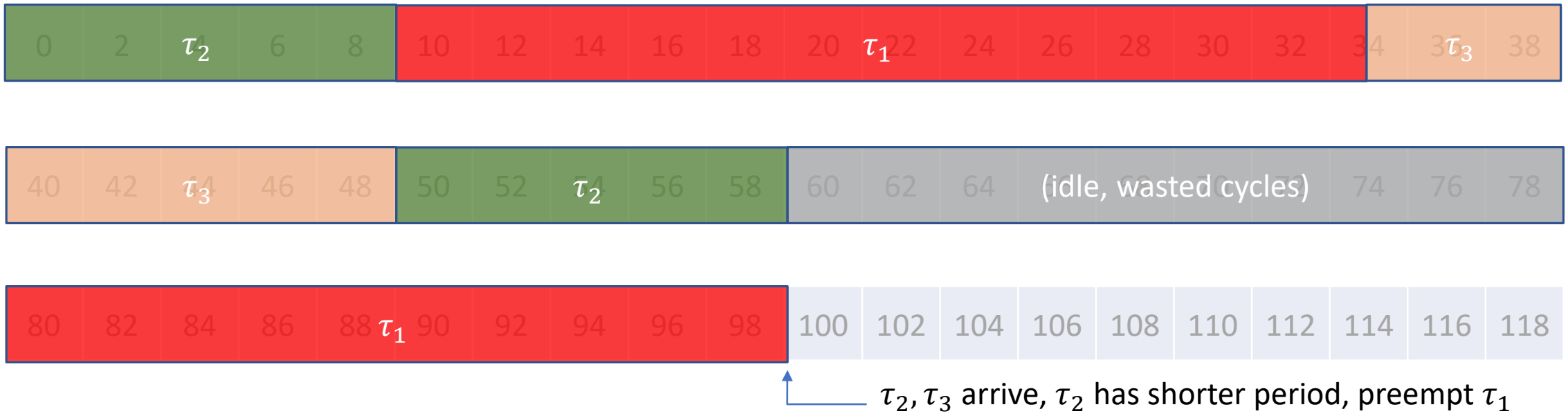
Run Queue at T = 80: $\{\tau_1\}$



Round Robin Scheduling

Task	Period	Arrival	Burst Length
τ_1	80	0	25
τ_2	50	0	10
τ_3	100	0	15

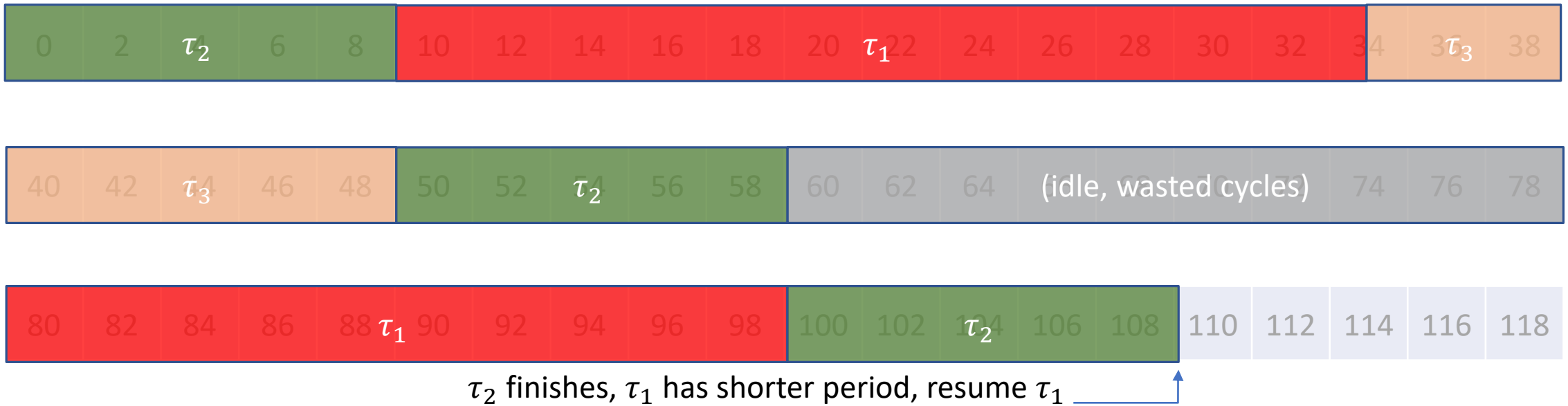
Run Queue at T = 100: $\{\tau_1, \tau_2, \tau_3\}$



Round Robin Scheduling

Task	Period	Arrival	Burst Length
τ_1	80	0	25
τ_2	50	0	10
τ_3	100	0	15

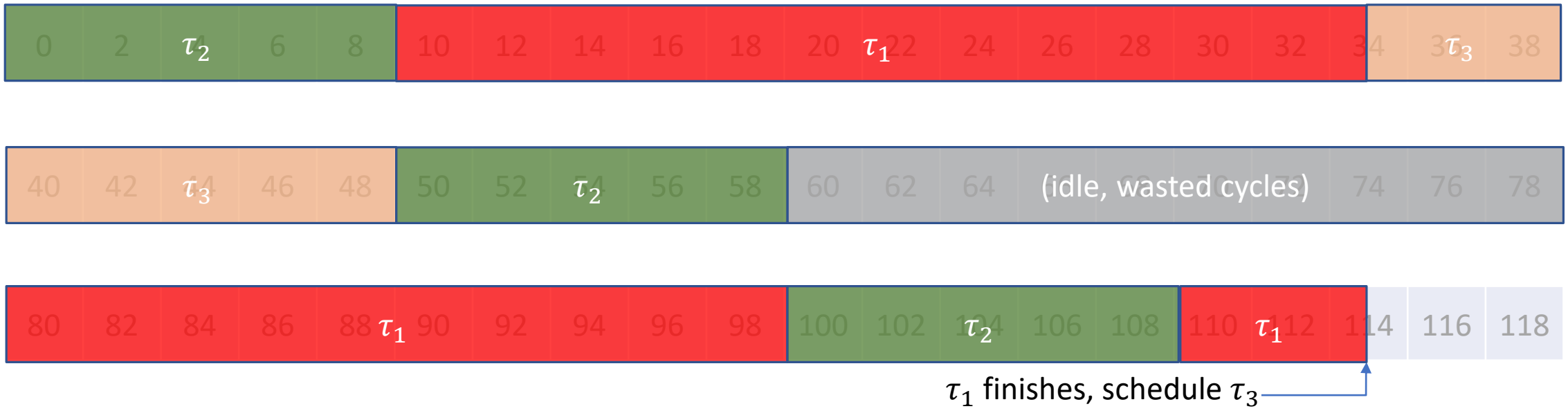
Run Queue at T = 110: $\{\tau_1, \tau_3\}$



Round Robin Scheduling

Task	Period	Arrival	Burst Length
τ_1	80	0	25
τ_2	50	0	10
τ_3	100	0	15

Run Queue at T = 115: $\{\tau_3\}$



CPU Usage

Task	Period	Arrival	Burst Length
τ_1	80	0	25
τ_2	50	0	10
τ_3	100	0	15

$$U = \frac{25}{80} + \frac{10}{50} + \frac{15}{100} = 66.25\%$$

CPU is idle 33.75% of the time.

Rate Monotonic Scheduling

Task	Period	Arrival	Burst Length
τ_1	80	0	20
τ_2	50	0	25
τ_3	100	0	15

0	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30	32	34	36	38
40	42	44	46	48	50	52	54	56	58	60	62	64	66	68	70	72	74	76	78
80	82	84	86	88	90	92	94	96	98	100	102	104	106	108	110	112	114	116	118

Rate Monotonic Scheduling

Task	Period	Arrival	Burst Length
τ_1	80	0	20
τ_2	50	0	25
τ_3	100	0	15

Run Queue at T = 0: $\{\tau_1, \tau_2, \tau_3\}$

0	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30	32	34	36	38

τ_1, τ_2, τ_3 arrive, τ_2 has shortest period thus is scheduled

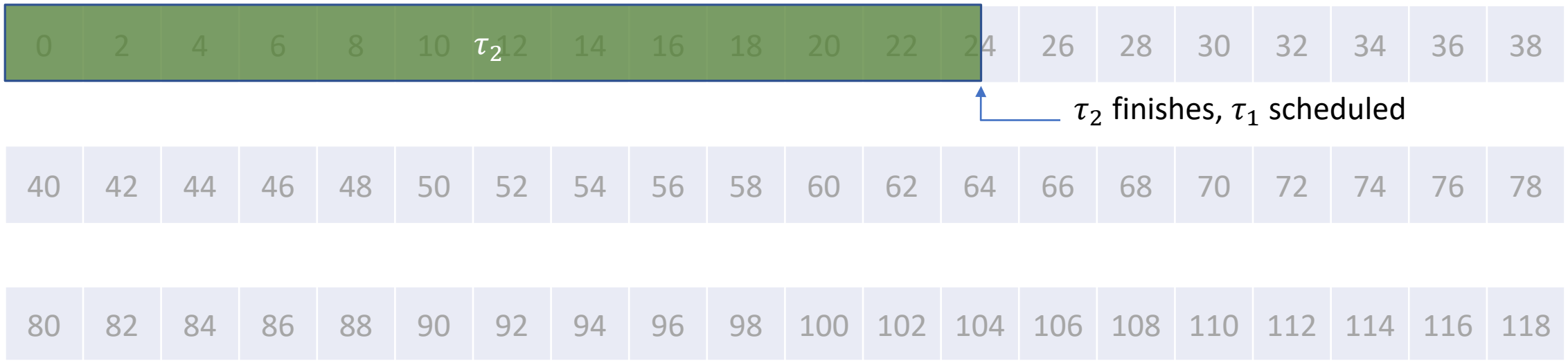
40	42	44	46	48	50	52	54	56	58	60	62	64	66	68	70	72	74	76	78

80	82	84	86	88	90	92	94	96	98	100	102	104	106	108	110	112	114	116	118

Rate Monotonic Scheduling

Task	Period	Arrival	Burst Length
τ_1	80	0	20
τ_2	50	0	25
τ_3	100	0	15

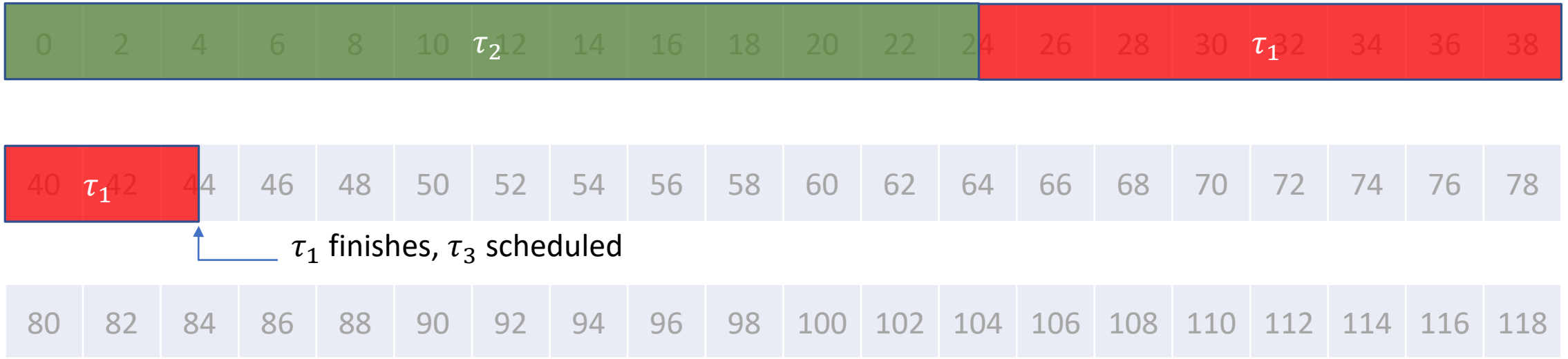
Run Queue at T = 25: $\{\tau_1, \tau_3\}$



Rate Monotonic Scheduling

Task	Period	Arrival	Burst Length
τ_1	80	0	20
τ_2	50	0	25
τ_3	100	0	15

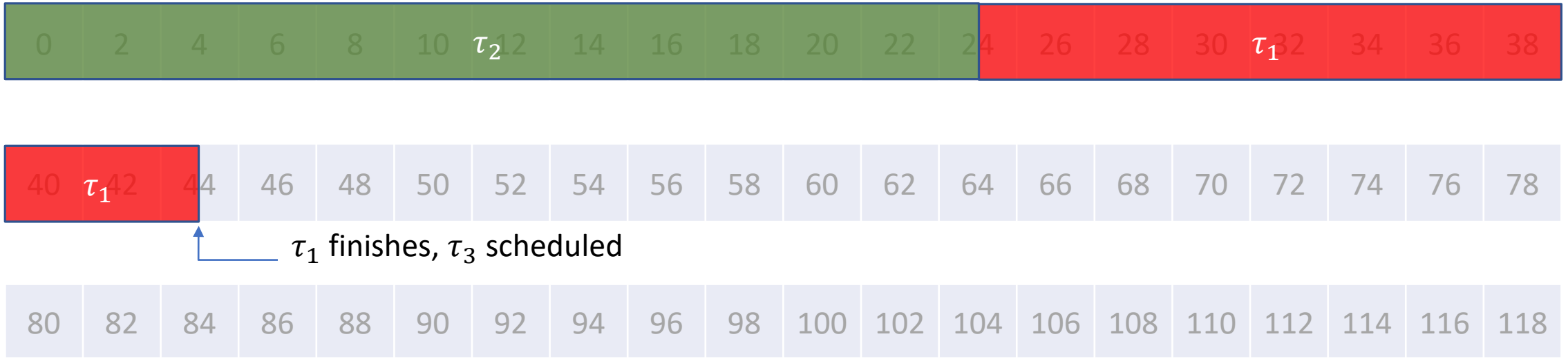
Run Queue at T = 45: $\{\tau_3\}$



Rate Monotonic Scheduling

Task	Period	Arrival	Burst Length
τ_1	80	0	20
τ_2	50	0	25
τ_3	100	0	15

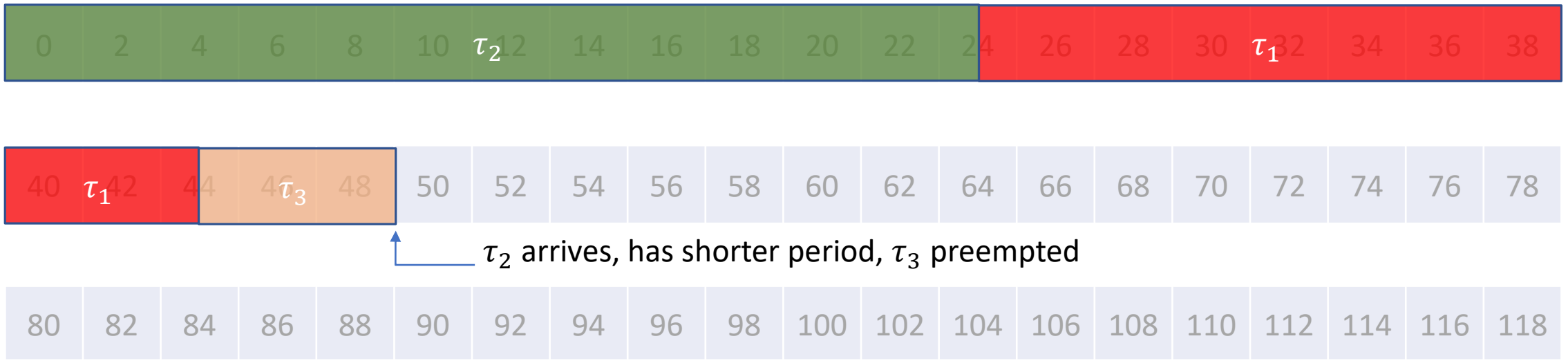
Run Queue at T = 45: $\{\tau_3\}$



Rate Monotonic Scheduling

Task	Period	Arrival	Burst Length
τ_1	80	0	20
τ_2	50	0	25
τ_3	100	0	15

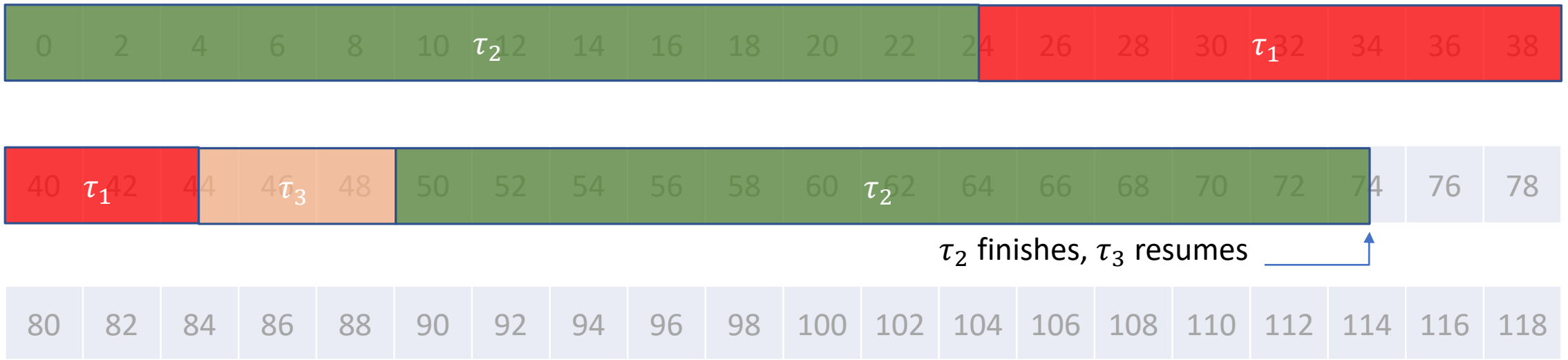
Run Queue at T = 50: $\{\tau_3, \tau_2\}$



Rate Monotonic Scheduling

Task	Period	Arrival	Burst Length
τ_1	80	0	20
τ_2	50	0	25
τ_3	100	0	15

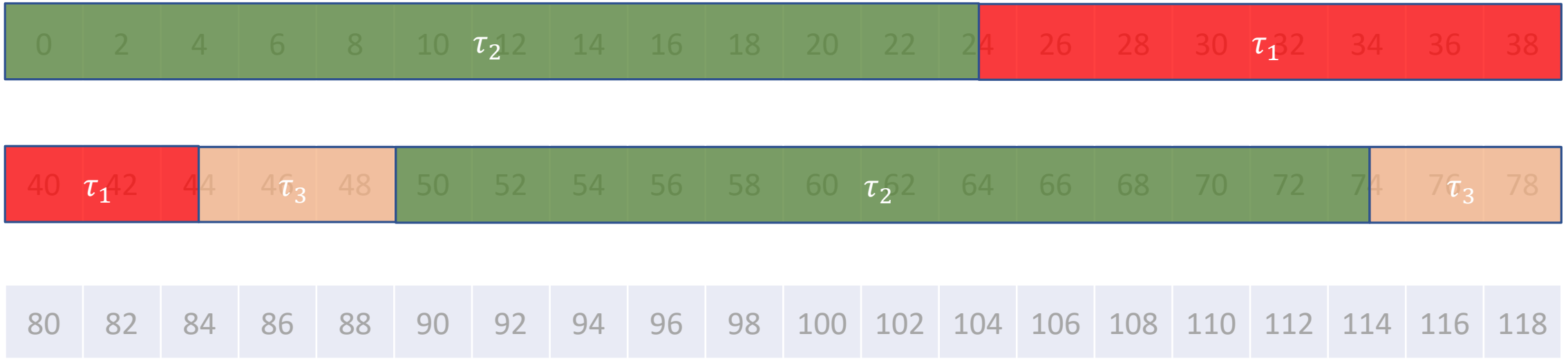
Run Queue at T = 75: $\{\tau_3\}$



Rate Monotonic Scheduling

Task	Period	Arrival	Burst Length
τ_1	80	0	20
τ_2	50	0	25
τ_3	100	0	15

Run Queue at T = 80: $\{\tau_3, \tau_1\}$

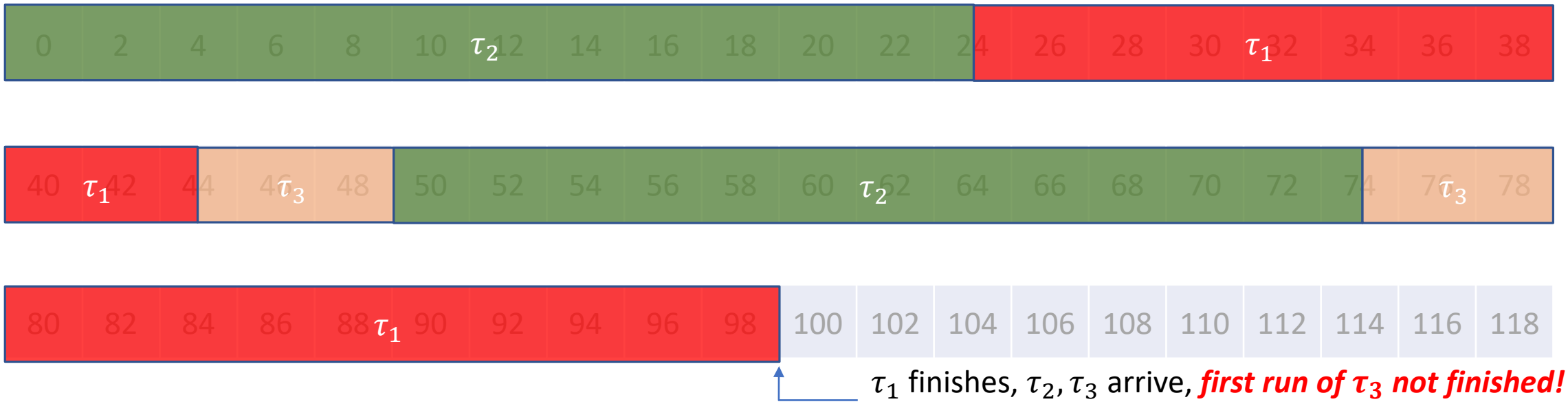


τ_1 arrives, has shorter period, τ_3 preempted

Rate Monotonic Scheduling

Task	Period	Arrival	Burst Length
τ_1	80	0	20
τ_2	50	0	25
τ_3	100	0	15

Run Queue at T = 80: $\{\tau_3, \tau_2, \boldsymbol{\tau_3}\}$



CPU Overcommitted?

Task	Period	Arrival	Burst Length
τ_1	80	0	20
τ_2	50	0	25
τ_3	100	0	15

$$U = \frac{20}{80} + \frac{25}{50} + \frac{15}{100} = 90\%$$

CPU Overcommitted?

Task	Period	Arrival	Burst Length
τ_1	80	0	20
τ_2	50	0	25
τ_3	100	0	15

$$U = \frac{20}{80} + \frac{25}{50} + \frac{15}{100} = 90\%$$

CPU is idle 10% of the time.

CPU Overcommitted?

Task	Period	Arrival	Burst Length
τ_1	80	0	20
τ_2	50	0	25
τ_3	100	0	15

$$U = \frac{20}{80} + \frac{25}{50} + \frac{15}{100} = 90\%$$

CPU is idle 10% of the time.

→ **Scheduler failed to meet deadlines.**

Rate Monotonic Scheduling

- Rate monotonic scheduler may fail if CPU utilization exceeds a threshold.

$$\sum_{i=1}^{n} \frac{C_i}{P_i} \leq n\left(2^{1/n} - 1\right)$$

Where *n* is the number of tasks [Liu & Layland, 1973].

- This condition is *sufficient*, but *not necessary*.

Rate Monotonic Scheduling

Task	Period	Arrival	Burst Length
τ_1	80	0	20
τ_2	50	0	25
τ_3	100	0	15

$$U = \frac{20}{80} + \frac{25}{50} + \frac{15}{100} = 90\%$$

Checking the condition:

$$3 \times \left(2^{1/3} - 1\right) \approx 78\% \ll U$$

Rate Monotonic Scheduling

Consider a set of *n* tasks $T = \{\tau_1, \tau_2, \dots, \tau_n\}$ whose execution periods and times, *P* and *C*, are $P = \{P_1, P_2, \dots, P_n\}$ and $C = \{C_1, C_2, \dots, C_n\}$, respectively. Suppose that task τ_i finishes executing at time t. Then, let

$$W_i(t) = \sum_{j=1}^{i} C_j \left\lceil \frac{t}{P_j} \right\rceil = t - idletime$$

$$L_i(t) = \frac{W_i(t)}{t}$$

$$L = \min_{0 \le t \le P_i} L_i(t)$$

Task τ_i can be feasibly scheduled using the rate monotonic algorithm if and only if $L_i(t) \le 1$. If so, then $\tau_1, \tau_2, \dots, \tau_{i-1}$ are also feasibly schedulable.

Rate Monotonic Scheduling

- Full schedulability test is solving the previous recurrent relation to determine violations
- [Liu & Layland 1973] also proved that under the assumption that
  - in an uniprocessor system we know the tasks periods and burst lengths and
  - that these are deterministic,
  - the tasks share no resources, and
  - that context switches have no impact

  rate monotonic scheduling is the *best* we can achieve.

Deadline Monotonic Scheduling

- Also known as *Earliest Deadline First* scheduling
- Real-time priority scheduler
- Attempt to overcome shortcomings of rate monotonic
  - Give priority to tasks that have earliest deadline
  - Higher priority tasks always preempt lower priority tasks

Deadline Monotonic Scheduling

Task	Period	Arrival	Burst Length
τ_1	80	0	20
τ_2	50	0	25
τ_3	100	0	15

0	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30	32	34	36	38
40	42	44	46	48	50	52	54	56	58	60	62	64	66	68	70	72	74	76	78
80	82	84	86	88	90	92	94	96	98	100	102	104	106	108	110	112	114	116	118

Deadline Monotonic Scheduling

Task	Period	Arrival	Burst Length
τ_1	80	0	20
τ_2	50	0	25
τ_3	100	0	15

Run Queue at T = 0: $\{\tau_1, \tau_2, \tau_3\}$

0	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30	32	34	36	38

τ_1, τ_2, τ_3 arrive, τ_2 earliest deadline so it is scheduled

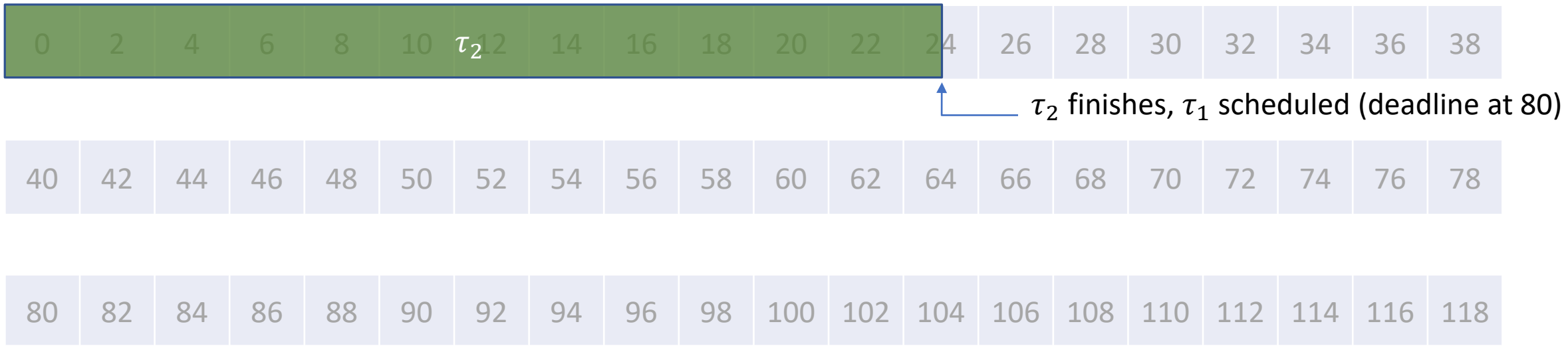
40	42	44	46	48	50	52	54	56	58	60	62	64	66	68	70	72	74	76	78

80	82	84	86	88	90	92	94	96	98	100	102	104	106	108	110	112	114	116	118

Deadline Monotonic Scheduling

Task	Period	Arrival	Burst Length
τ_1	80	0	20
τ_2	50	0	25
τ_3	100	0	15

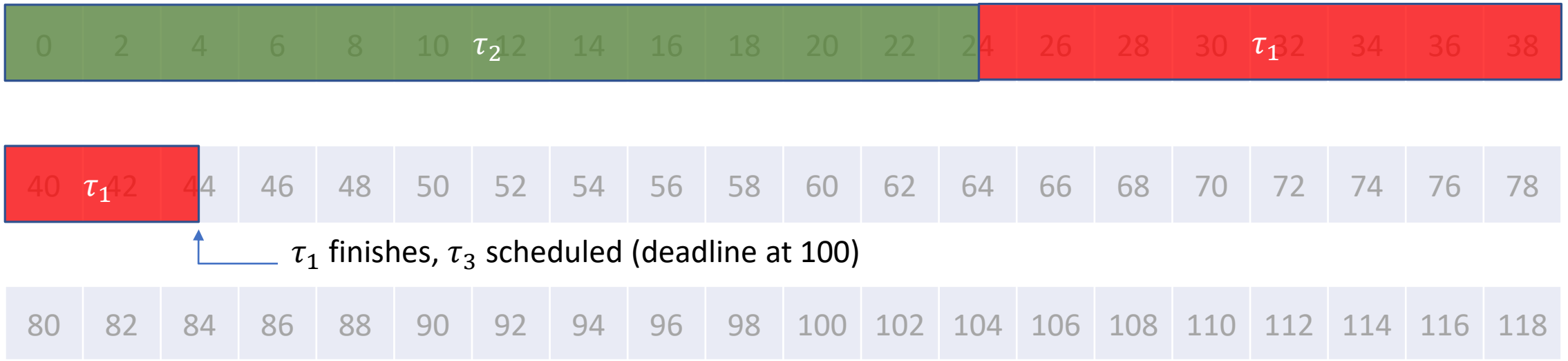
Run Queue at T = 25: $\{\tau_1, \tau_3\}$



Deadline Monotonic Scheduling

Task	Period	Arrival	Burst Length
τ_1	80	0	20
τ_2	50	0	25
τ_3	100	0	15

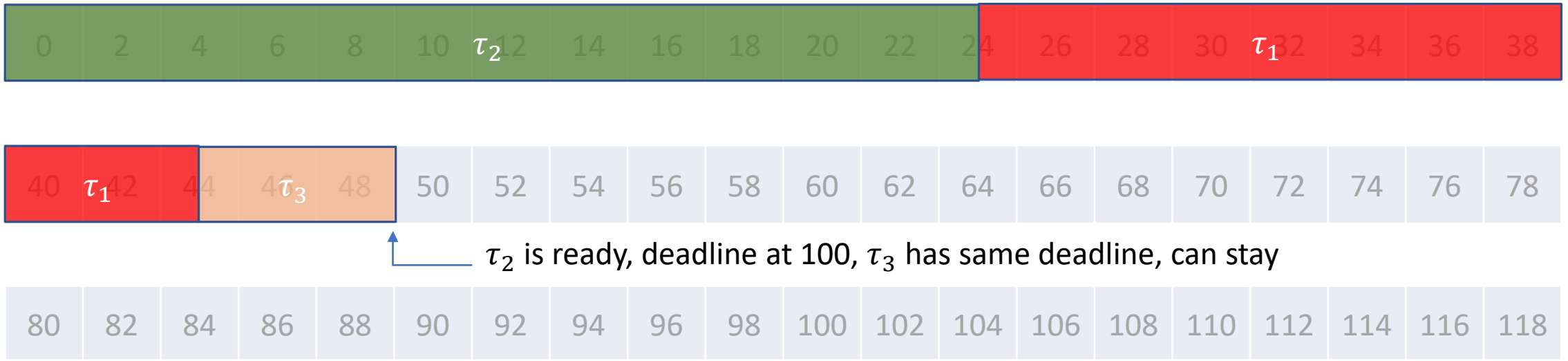
Run Queue at T = 45: $\{\tau_3\}$



Deadline Monotonic Scheduling

Task	Period	Arrival	Burst Length
τ_1	80	0	20
τ_2	50	0	25
τ_3	100	0	15

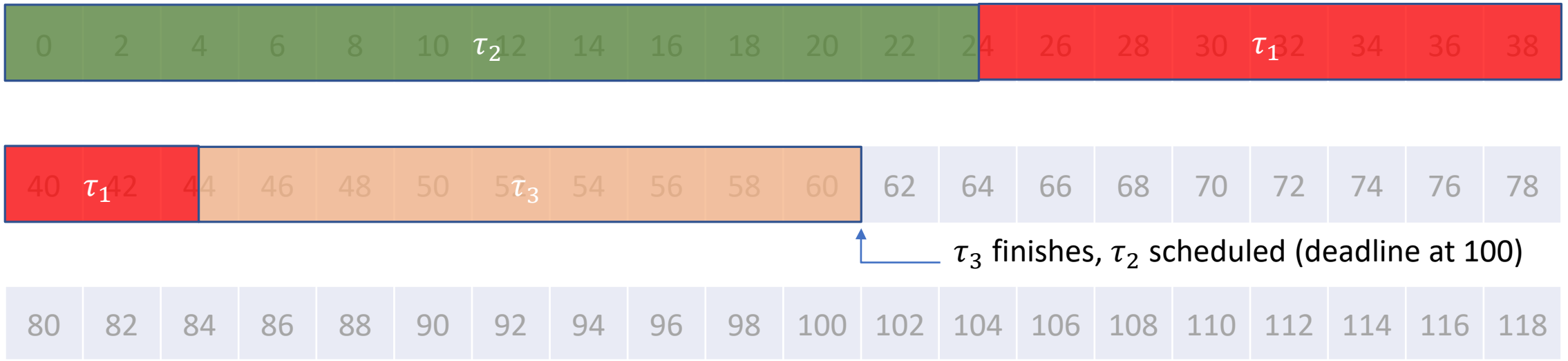
Run Queue at T = 50: $\{\tau_3, \tau_2\}$



Deadline Monotonic Scheduling

Task	Period	Arrival	Burst Length
τ_1	80	0	20
τ_2	50	0	25
τ_3	100	0	15

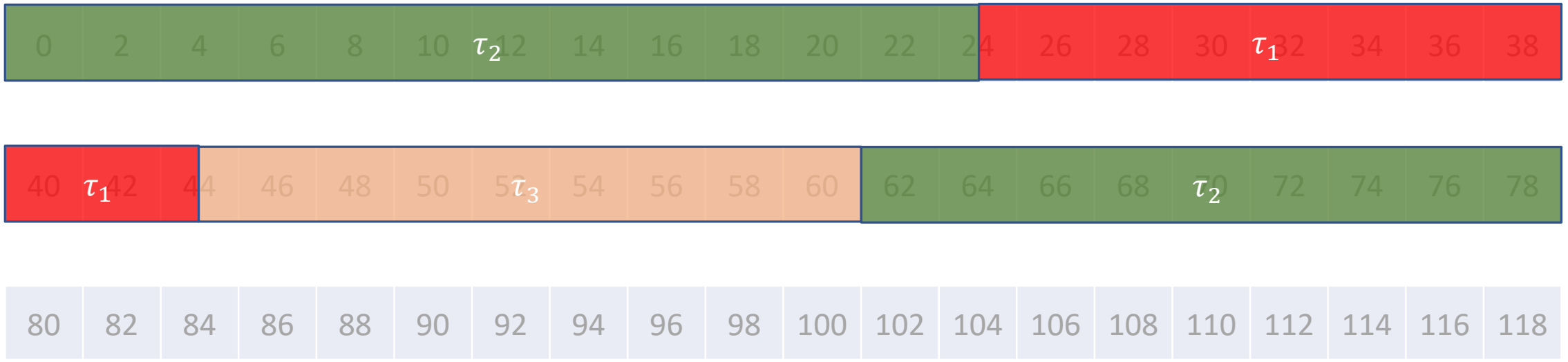
Run Queue at T = 60: $\{\tau_2\}$



Deadline Monotonic Scheduling

Task	Period	Arrival	Burst Length
τ_1	80	0	20
τ_2	50	0	25
τ_3	100	0	15

Run Queue at T = 80: $\{\tau_2, \tau_1\}$



τ_1 arrives (deadline at 160), τ_2 remains (deadline at 100)

Deadline Monotonic Scheduling

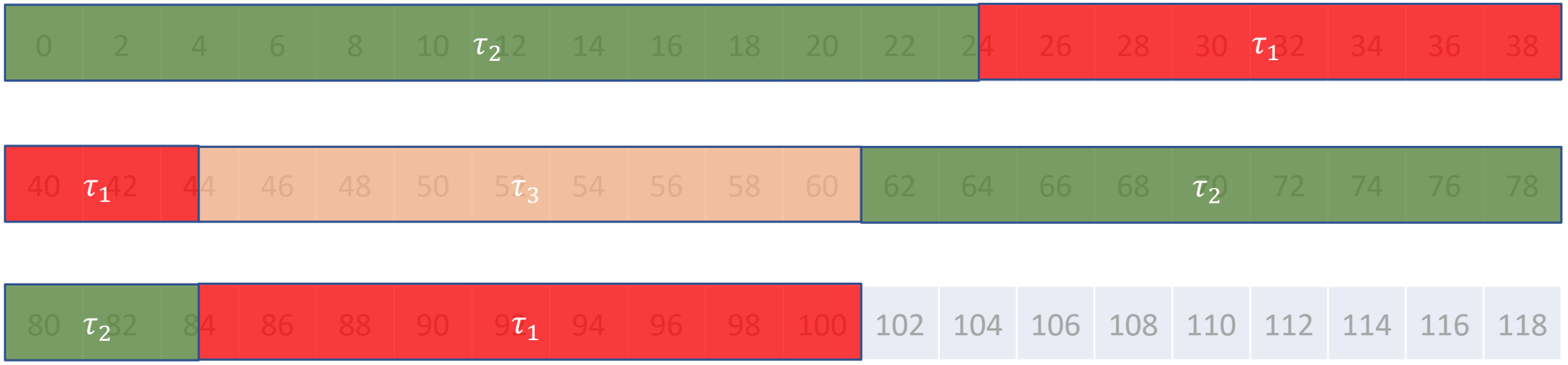
Task	Period	Arrival	Burst Length
τ_1	80	0	20
τ_2	50	0	25
τ_3	100	0	15

Run Queue at T = 85: $\{\tau_1\}$

Deadline Monotonic Scheduling

Task	Period	Arrival	Burst Length
τ_1	80	0	20
τ_2	50	0	25
τ_3	100	0	15

Run Queue at T = 100: $\{\tau_1, \tau_2, \tau_3\}$



Deadline Monotonic Scheduling

Under this algorithm, if all tasks are periodic and have relative deadlines equal to their periods, they can be feasibly scheduled if and only if

$$\sum_{i=1}^{n} \frac{C_i}{P_i} \leq 1$$

There is no simple schedulability test corresponding to the case where the relative deadlines do not equal the periods. In such case, we actually have to develop a schedule using the deadline monotonic algorithm to see if all deadlines are met over a given time interval.

Implementing These Schedulers

Use a priority queue:

- For Rate Monotonic Scheduler, priority is given to the shortest task
- For Deadline Monotonic Scheduler, priority is given to the earliest deadline

Recall that:

- Tasks must be periodic
- Tasks must declare how long they will run for
- Tasks must declare when their deadline is