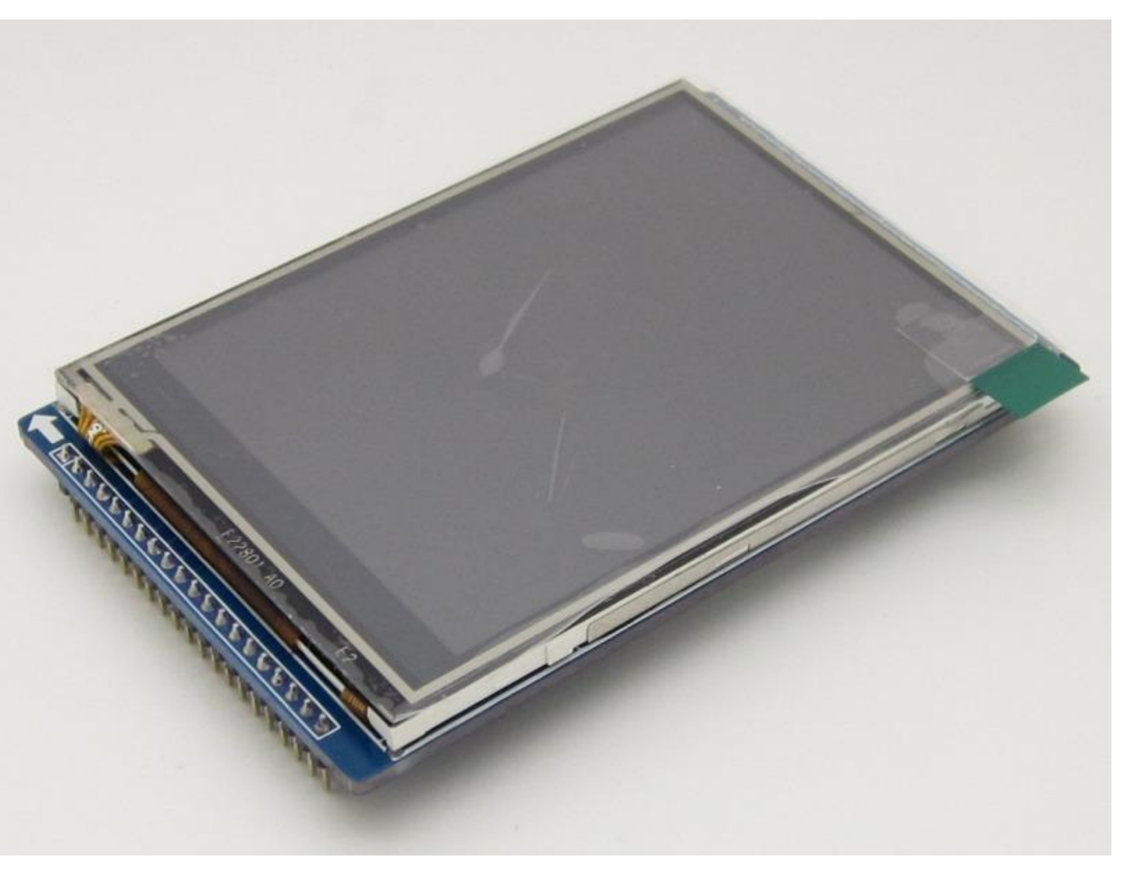
Resistive Touch Screen Liquid Crystal Display Module

THREAD PRIORITY, PRIORITY INVERSION, PRIORITY INHERITANCE, APERIODIC EVENTS

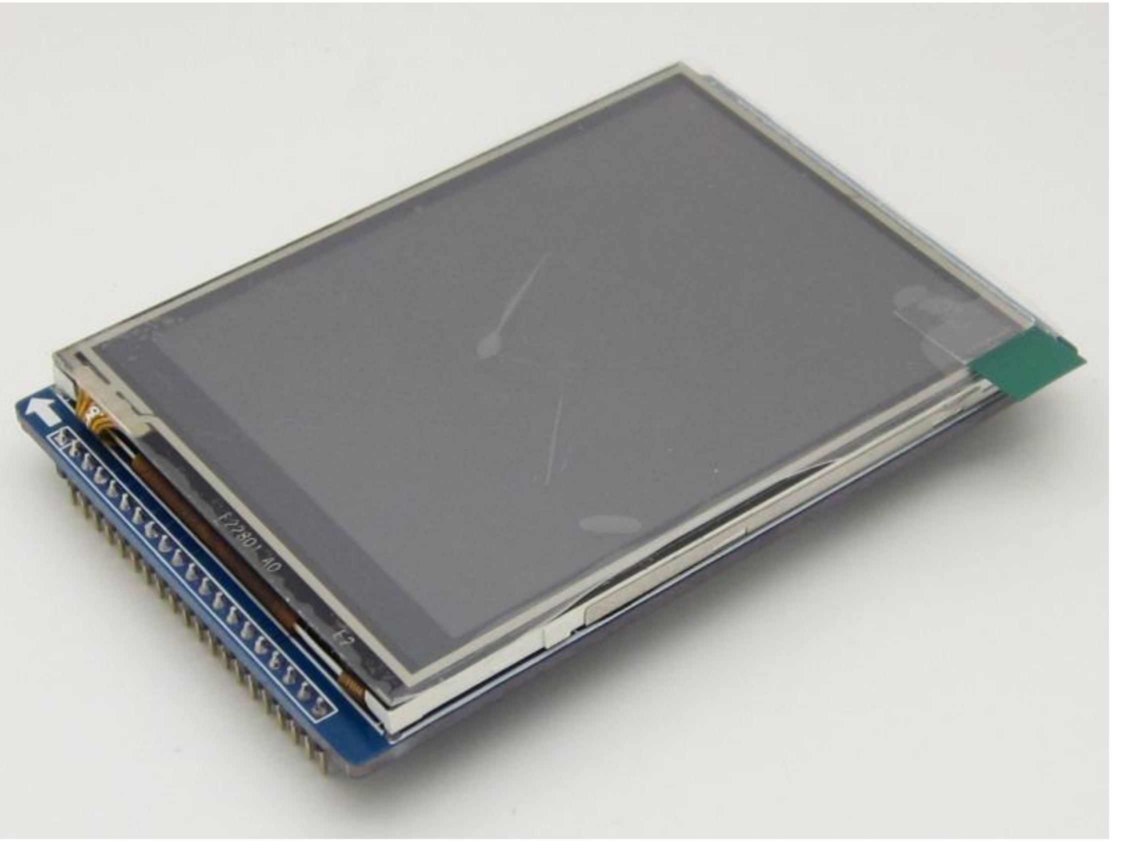
2.8'' TFT LCD Module

- Model HY28B
- LCD Controller: ILI9325
- Touchscreen Controller: ADS7843
- Interface:
  - SPI
  - 8-bit parallel
  - 16-bit parallel
- Touch screen interface:
  - SPI



2.8'' TFT LCD Module

- Colors: 65536
- Resolution: 320 by 240



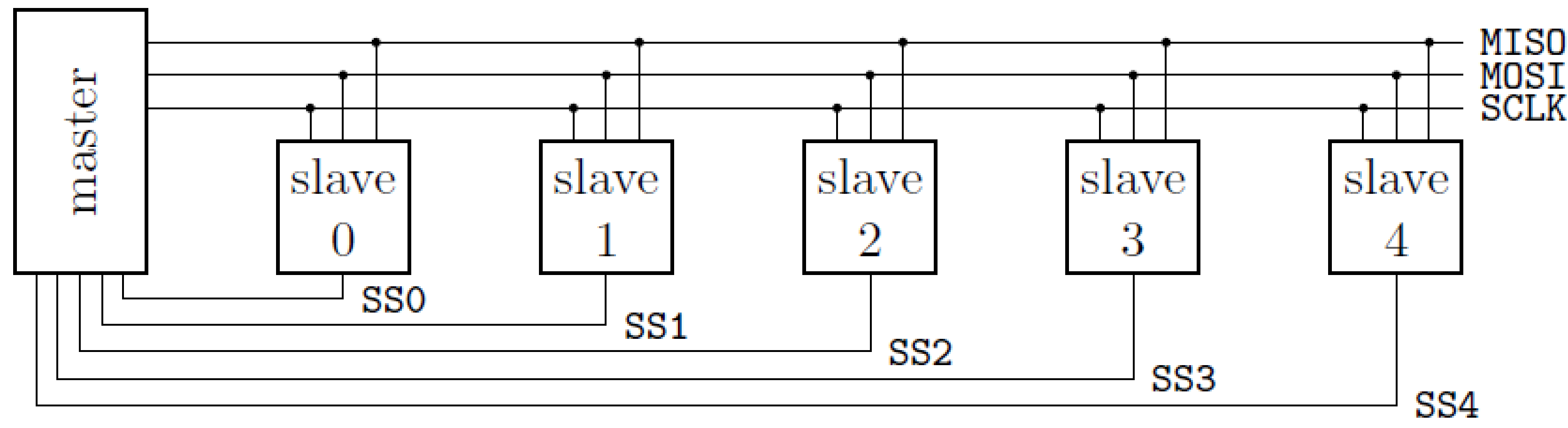
Serial Peripheral Interface

- Three main signals
  - SCLK: Serial Clock
  - MOSI: Master Out Slave In
  - MISO: Master In Slave Out
- Allows multiple devices to reside on the same bus
  - CS: Chip Select (sometimes SS: Slave Select)
  - Must have a number of CS signals equal to number of slaves on the topology
  - Only one CS line can be asserted at a time
  - Master is responsible for asserting CS

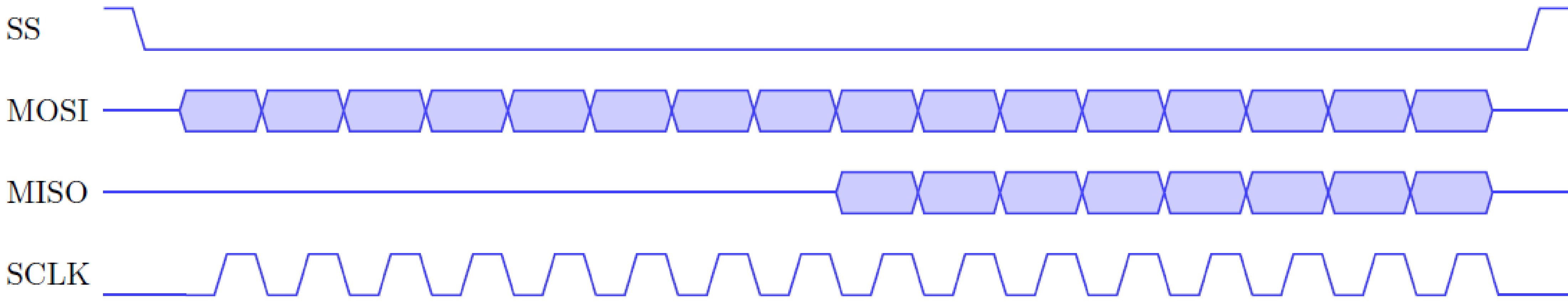
Serial Peripheral Interface

- SCLK: Serial Clock
  - Always driven by master
  - Rising edge of clock shifts data from master and slave device
  - Falling edge of clock samples data on master and slave device
- MOSI: Master Out Slave In
  - Used by master to transmit data to the asserted slave
  - Slaves receivers are connected to this line
- MISO: Master In Slave Out
  - Used by master to receive data from the asserted slave
  - Slave transmitters are connected to this line

Serial Peripheral Interface



Serial Peripheral Interface

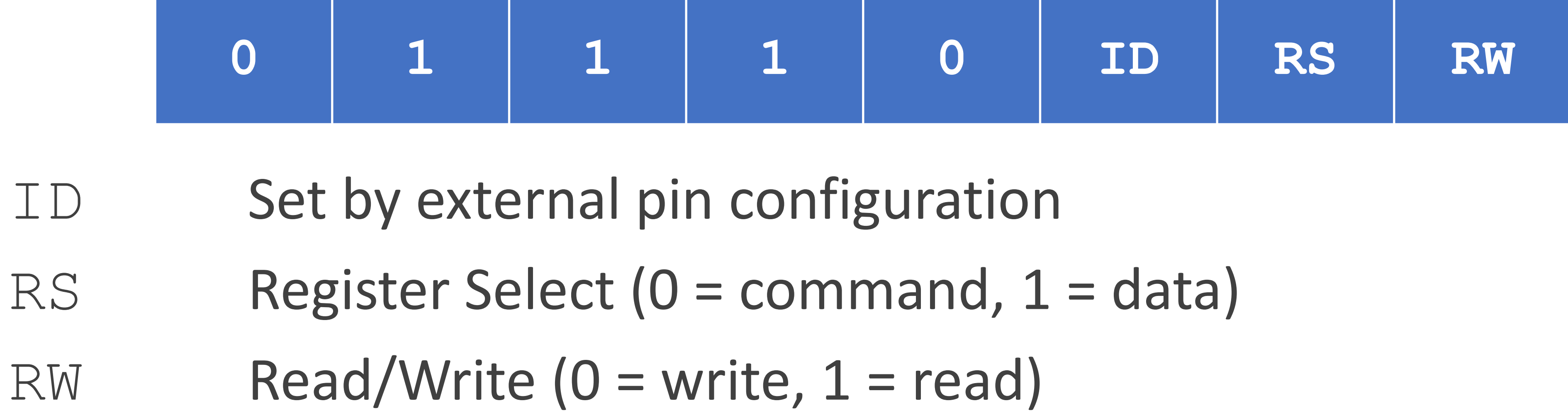


LCD Operation

- Command interface
- LCD controller will receive commands from one of the three [selectable] interfaces
- LCD controller will execute the command
  - Move cursor
  - Set pixel color

LCD Operation

- SPI interface requires clock to idle on HIGH
- SPI interface requires sending a start byte



- After start byte is sent, data and commands can be sent to LCD

LCD Example SPI Frame

Start
End

nCS (Input)

SCL (Input)

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24

SDI (Input)

0 1 1 1 0 ID RS RW D15 D14 D13 D12 D11 D10 D9 D8 D7 D6 D5 D4 D3 D2 D1 D0

Start Byte

Index register, registers setting, and GRAM write

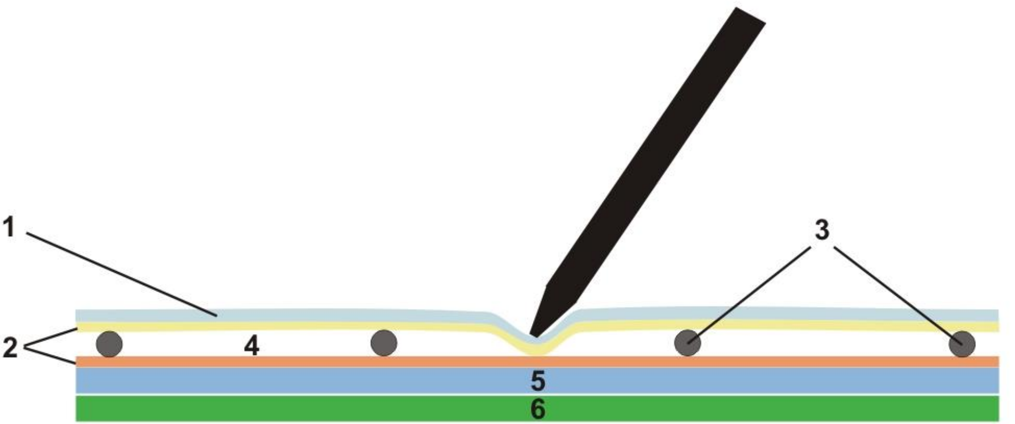
SDO (Output)

D15 D14 D13 D12 D11 D10 D9 D8 D7 D6 D5 D4 D3 D2 D1 D0

Status, registers read and GRAM read

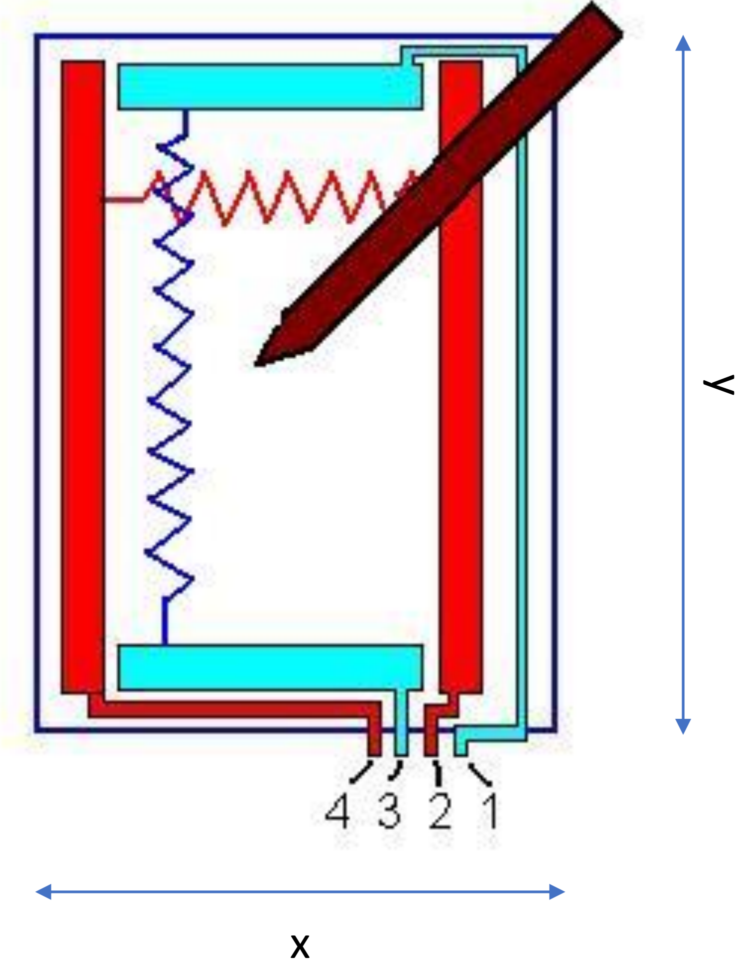
Touch Screen Functionality

- Resistive touchscreen
  1. Transparent film
  2. ITO conductive coating
  3. Spacer dots
  4. Airgap
  5. Bottom circuit layer
  6. Backing panel



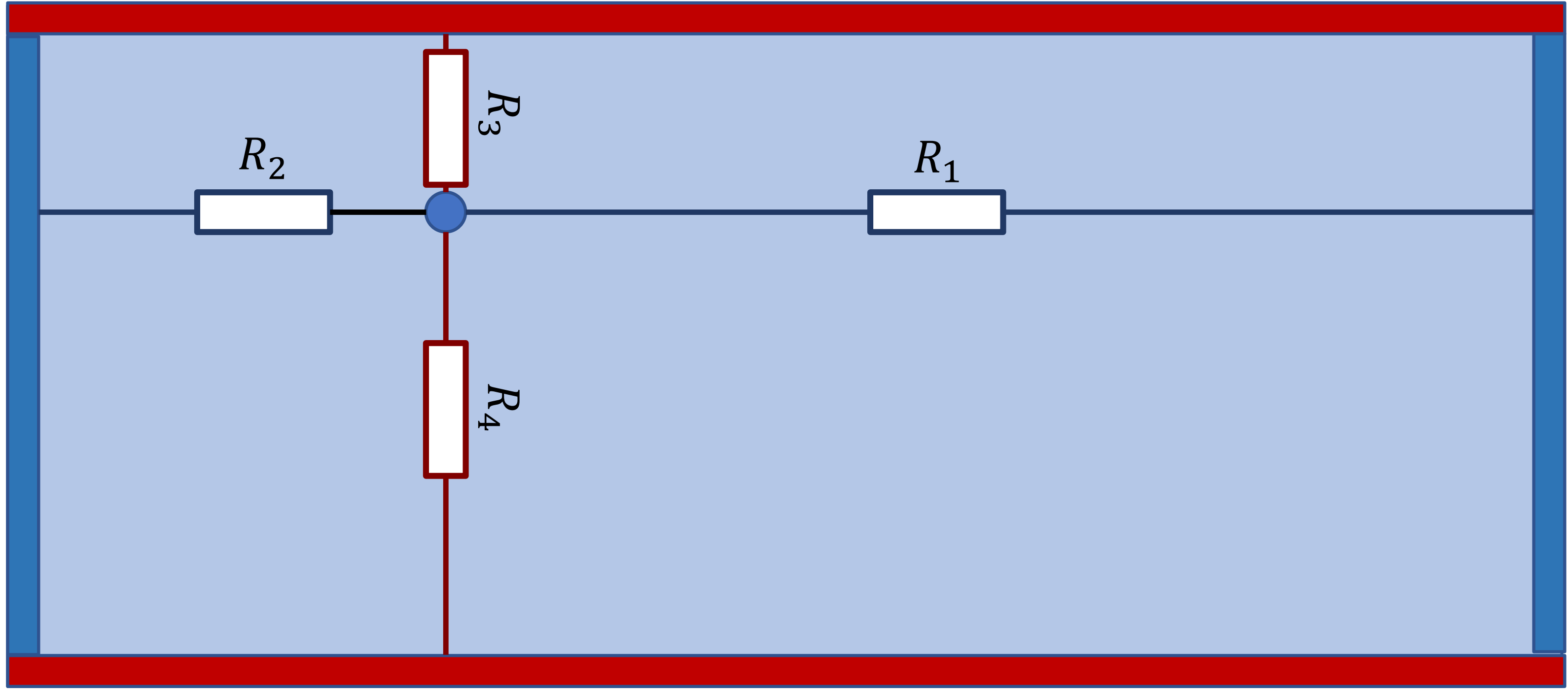
Touch Screen Functionality

- When pressed a resistor network is created
- Controller measures voltage created in voltage divider
  - Get X: set known voltages in 2 and 4 rails, measure voltage in 1 or 3
  - Get Y: set known voltages in 1 and 3 rails, measure voltage in 2 or 4



Touch Screen Functionality

Circuit on touch:



Touch Screen Functionality

Get Y coordinate:

R_3

R_2

R_1

V_O

V_y

R_4

Touch Screen Functionality

Y-coordinate is determined by measuring V_0

$$V_0 = \frac{R_i R_4}{R_2(R_3 + R_4) + R_4 R_3 + R_i(R_4 + R_3)} V_y$$

ADC input is high impedance:

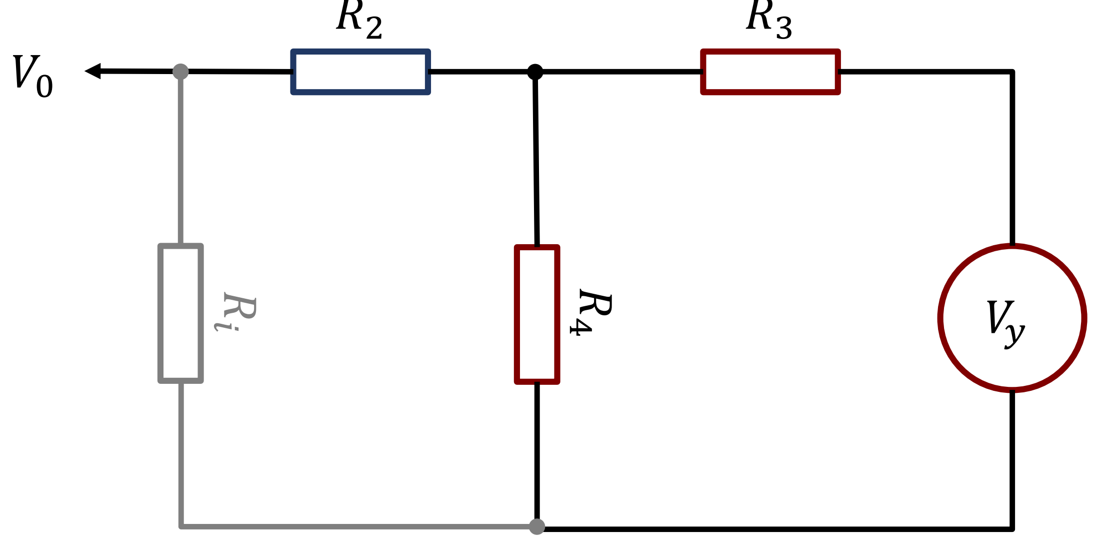
$$R_i \gg R_2, R_3, R_4$$

Then:

$$V_0 \approx \frac{R_4}{R_4 + R_3} V_y$$

To determine position, find R_4:

$$R_4 = (R_4 + R_3) \frac{V_0}{V_y}$$



R_i is the ADC's input resistance

Touch Screen Functionality

- The ADS7843 performs these actions for you
  - 12-bit sampling ADC
  - Low output (source) resistance (5-6 Ω)
  - High input impedance (5 GΩ)
  - Serial interface (Similar to SPI but not quite like it)
- You are still responsible for converting the analog voltage value into a position value

ADS7843 Operation

- First 8 clock cycles: send control byte

S	A2	A1	A0	MODE	SER/~DFR	PD1	PD0

- Wait 1 clock cycles
- Next 12 clock cycles are data
- Fill rest with 3 clocks (DOUT=0)

→ 24 clock cycles total

Bit	Description
S	Start bit. Always 1.
A2-A0	Channel Select bits.
MODE	12-bit (0)/8-bit (1) conversion bit.
SER/~DFR	Single-ended/differential reference select bit.
PD1-PD0	Power-down mode select bits.

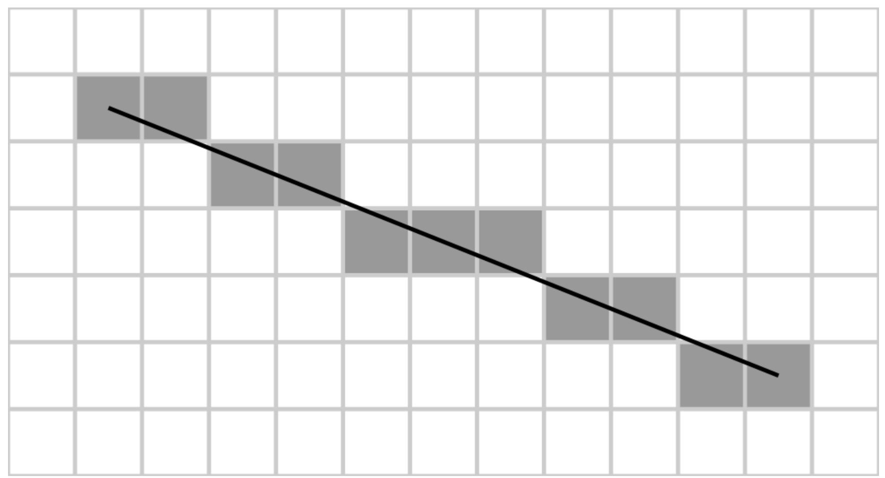
ADS7843 Operation

CS
DCLK
t_{ACQ}
1 8 1 8 1 8
DIN
S A2 A1 A0 MODE SER/DFR PD1 PD0
(START)
Idle
Acquire
Conversion
Idle
BUSY
DOUT
11 10 9 8 7 6 5 4 3 2 1 0
Zero Filled...
(MSB)
(LSB)
X/Y SWITCHES(1)
(SER/DFR HIGH)
OFF ON OFF
X/Y SWITCHES(1, 2)
(SER/DFR LOW)
OFF ON OFF

NOTES: (1) Y Drivers are on when X+ is selected input channel (A2-A0 = 001_B), X Drivers are on when Y+ is selected input channel (A2-A0 = 101_B). Y– will turn on when power-down mode is entered and PD1, PD0 = 00_B. (2) Drivers will remain on if power-down mode is 11_B (no power-down) until selected input channel, reference mode, or power-down mode is changed.

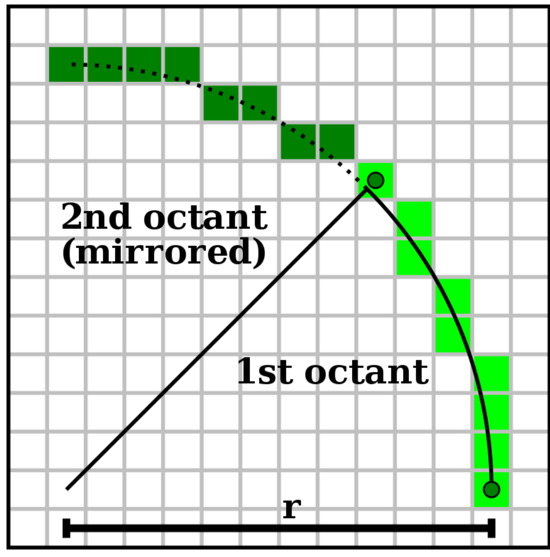
Drawing Graphics

- Bresenham’s Line Algorithm
  - Avoid expensive floating point computations when drawing a slanted line in a pixel matrix
  - Incremental error algorithm
  - Developed at IBM in 1962
  - https://en.wikipedia.org/wiki/Bresenham%27s_line_algorithm#All_cases



Drawing Graphics

- Midpoint Circle Algorithm
  - Draw a circle in a pixel matrix without using floating point computations
  - Similar to Bresenham's Line Algorithm
  - Divides circle into 8 octants, draws them simultaneously
  - https://en.wikipedia.org/wiki/Midpoint_circle_algorithm#C_example



Implementation Required

- Provided Functions:

Function	Description
`LCD_Init()`	Initializes the LCD
`PutChar()`	Writes a character on the LCD
`LCD_Text()`	Writes a string on the LCD
`LCD_WriteIndex()`	Sets address for register to write
`LCD_WriteData()`	Writes data to specified register
`LCD_ReadData()`	Reads data from specified index
`LCD_Write_Data_Start()`	Sends starting condition for continuous data transmission.

Priority Scheduling

THREAD PRIORITY, PRIORITY INVERSION, PRIORITY INHERITANCE, APERIODIC EVENTS

Scheduler Recap

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

Given a set of tasks $T = \{\tau_1, \tau_2, \ldots, \tau_n\}$*, a set of processors* $\pi = \{\pi_1, \pi_2, \ldots, \pi_m\}$*, and a set of resources* $R = \{R_1, R_2, \ldots, R_k\}$*,* 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.*

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…

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

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

Thread Priority

- So far we have looked at two different priority scheduling mechanisms
  - Rate Monotonic Scheduling: shortest period → highest priority
  - Deadline Monotonic Scheduling: earliest deadline → highest priority
- These scheduling mechanisms are for periodic tasks
  - Recall that periodic tasks have a start time and an end time
- What about priority scheduling for ``always on'' tasks?
- What about priority scheduling for longer but important tasks?

Thread Priority

- Round robin scheduler as implemented has no priority
  - If task is runnable, it can be scheduled
  - A runnable task may need to perform certain actions at a higher priority
  - Scheduler must be modified to account for this!
- Deadline Monotonic Scheduling will use task deadlines as priority
  - What about tasks with long deadlines but that perform critical functionality?

Example Scenario

- Data acquisition loop
  - Captures data from an external sensor (e.g. temperature)
- Data logging loop
  - Logs acquired data

Which task should have higher priority? Why?

Example Scenario – Round Robin

- Give priority to logger
  - Assumes data has been acquired, ready to be recorded
  - Possible to miss critical data points
- Give priority to data acquisition
  - Assumes data can be logged as fast as being acquired
  - No missed data points
  - Insight: time of logging does not matter as long as data is logged

Possibility: Give two quantums to data acquisition and one quantum to logger.

Priority and Shared Resources

Task	Priority	Deadline	Arrival	Burst Length
τ_1	Highest	80	0	40
τ_2	Lowest	90	10	30

One CPU Quantum:

10 ticks

Tasks τ_1, τ_2 have a shared resource, will attempt to acquire it 5 ticks from their start, and relinquish it after 6 ticks after acquisition. Highest, medium, and lowest priority tasks will receive 3, 2, and 1 continuous CPU quantums, respectively.

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

Priority and Shared Resources

Task	Priority	Deadline	Arrival	Burst Length
τ_1	Highest	80	0	40
τ_2	Lowest	90	10	30

One CPU Quantum:

10 ticks

Tasks τ_1, τ_2 have a shared resource, will attempt to acquire it 5 ticks from their start, and relinquish it after 6 ticks after acquisition. Highest, medium, and lowest priority tasks will receive 3, 2, and 1 continuous CPU quantums, respectively.

0 τ_1	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30	32	34	36	38

τ_1 acquires lock for shared resource

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

Priority and Shared Resources

Task	Priority	Deadline	Arrival	Burst Length
τ_1	Highest	80	0	40
τ_2	Lowest	90	10	30

One CPU Quantum:

10 ticks

Tasks τ_1, τ_2 have a shared resource, will attempt to acquire it 5 ticks from their start, and relinquish it after 6 ticks after acquisition. Highest, medium, and lowest priority tasks will receive 3, 2, and 1 continuous CPU quantums, respectively.

0	2	τ_1	6	8	10	12	14	16	18	20	22	24	26	28	30	32	34	36	38

quantum expires, τ_2 arrives, τ_1 continues [highest priority]

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

Priority and Shared Resources

Task	Priority	Deadline	Arrival	Burst Length
τ_1	Highest	80	0	40
τ_2	Lowest	90	10	30

One CPU Quantum:

10 ticks

Tasks τ_1, τ_2 have a shared resource, will attempt to acquire it 5 ticks from their start, and relinquish it after 6 ticks after acquisition. Highest, medium, and lowest priority tasks will receive 3, 2, and 1 continuous CPU quantums, respectively.

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

τ_1 (0–10)

lock released

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

Priority and Shared Resources

Task	Priority	Deadline	Arrival	Burst Length
τ_1	Highest	80	0	40
τ_2	Lowest	90	10	30

One CPU Quantum:

10 ticks

Tasks τ_1, τ_2 have a shared resource, will attempt to acquire it 5 ticks from their start, and relinquish it after 6 ticks after acquisition. Highest, medium, and lowest priority tasks will receive 3, 2, and 1 continuous CPU quantums, respectively.

0	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30	32	34	36	38
τ_1																			

second quantum elapsed

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

Priority and Shared Resources

Task	Priority	Deadline	Arrival	Burst Length
τ_1	Highest	80	0	40
τ_2	Lowest	90	10	30

One CPU Quantum:

10 ticks

Tasks τ_1, τ_2 have a shared resource, will attempt to acquire it 5 ticks from their start, and relinquish it after 6 ticks after acquisition. Highest, medium, and lowest priority tasks will receive 3, 2, and 1 continuous CPU quantums, respectively.

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

τ_1 (0–28)

third quantum elapsed, schedule τ_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

Priority and Shared Resources

Task	Priority	Deadline	Arrival	Burst Length
τ_1	Highest	80	0	40
τ_2	Lowest	90	10	30

One CPU Quantum:

10 ticks

Tasks τ_1, τ_2 have a shared resource, will attempt to acquire it 5 ticks from their start, and relinquish it after 6 ticks after acquisition. Highest, medium, and lowest priority tasks will receive 3, 2, and 1 continuous CPU quantums, respectively.

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

τ_1 (0–30) τ_2 (30–35)

τ_2 acquires lock for shared resource

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

Priority and Shared Resources

Task	Priority	Deadline	Arrival	Burst Length
τ_1	Highest	80	0	40
τ_2	Lowest	90	10	30

One CPU Quantum:
10 ticks

Tasks τ_1, τ_2 have a shared resource, will attempt to acquire it 5 ticks from their start, and relinquish it after 6 ticks after acquisition. Highest, medium, and lowest priority tasks will receive 3, 2, and 1 continuous CPU quantums, respectively.

0	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30	32	34	36	38
							τ_1										τ_2		

quantum elapses, schedule τ_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

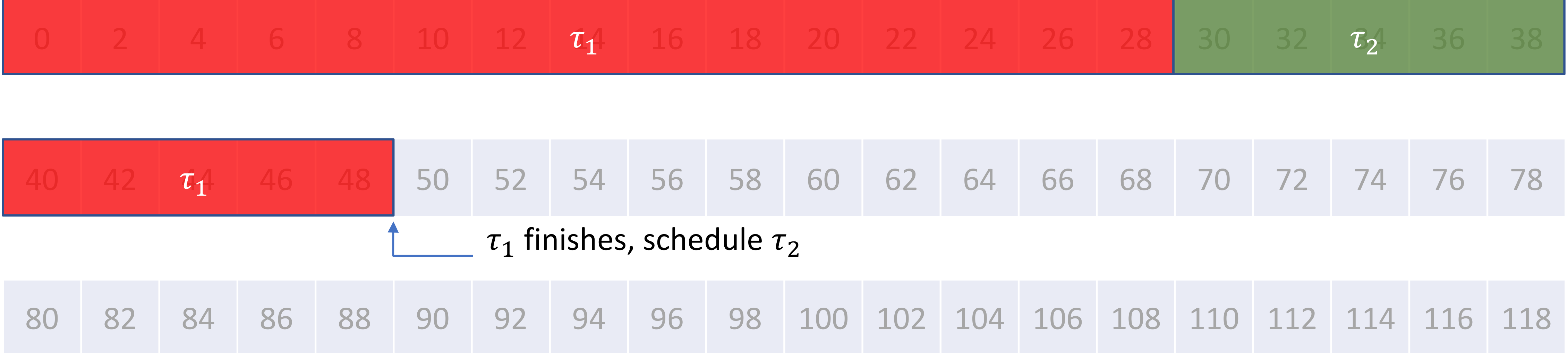
Priority and Shared Resources

Task	Priority	Deadline	Arrival	Burst Length
τ_1	Highest	80	0	40
τ_2	Lowest	90	10	30

One CPU Quantum:

10 ticks

Tasks τ_1, τ_2 have a shared resource, will attempt to acquire it 5 ticks from their start, and relinquish it after 6 ticks after acquisition. Highest, medium, and lowest priority tasks will receive 3, 2, and 1 continuous CPU quantums, respectively.



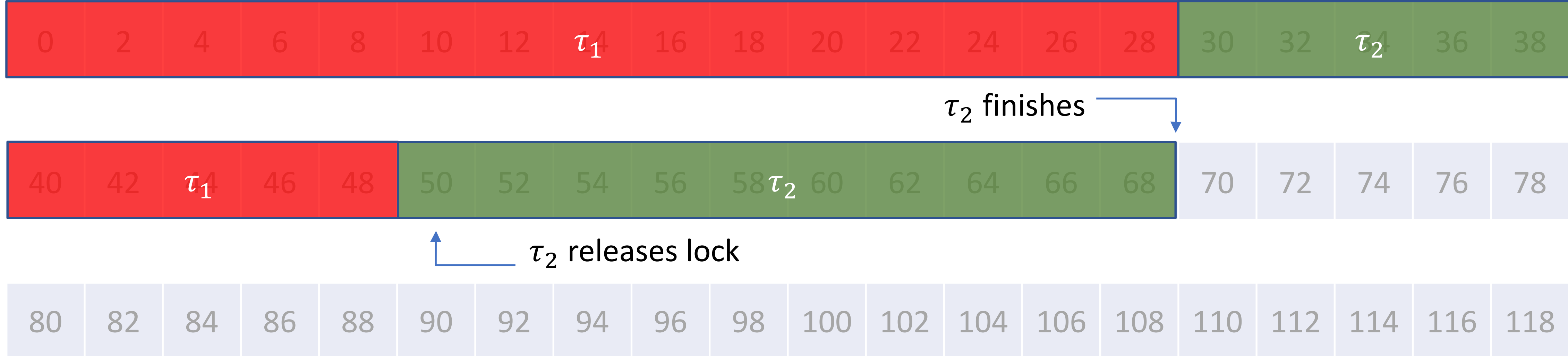
Priority and Shared Resources

Task	Priority	Deadline	Arrival	Burst Length
τ_1	Highest	80	0	40
τ_2	Lowest	90	10	30

One CPU Quantum:

10 ticks

Tasks τ_1, τ_2 have a shared resource, will attempt to acquire it 5 ticks from their start, and relinquish it after 6 ticks after acquisition. Highest, medium, and lowest priority tasks will receive 3, 2, and 1 continuous CPU quantums, respectively.



Priority and Shared Resources

Task	Priority	Deadline	Arrival	Burst Length
τ_1	Highest	80	0	40
τ_2	Lowest	90	10	30

One CPU Quantum:

10 ticks

Tasks τ_1, τ_2 have a shared resource, will attempt to acquire it 5 ticks from their start, and relinquish it after 6 ticks after acquisition. Highest, medium, and lowest priority tasks will receive 3, 2, and 1 continuous CPU quantums, respectively.

Priority and Shared Resources

Task	Priority	Deadline	Arrival	Burst Length
τ_1	Highest	80	6	40
τ_2	Lowest	90	0	30

One CPU Quantum:

10 ticks

Tasks τ_1, τ_2 have a shared resource, will attempt to acquire it 5 ticks from their start, and relinquish it after 6 ticks after acquisition. Highest, medium, and lowest priority tasks will receive 3, 2, and 1 continuous CPU quantums, respectively.

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

Priority and Shared Resources

Task	Priority	Deadline	Arrival	Burst Length
τ_1	Highest	80	6	40
τ_2	Lowest	90	0	30

One CPU Quantum:

10 ticks

Tasks τ_1, τ_2 have a shared resource, will attempt to acquire it 5 ticks from their start, and relinquish it after 6 ticks after acquisition. Highest, medium, and lowest priority tasks will receive 3, 2, and 1 continuous CPU quantums, respectively.

τ_2 0	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30	32	34	36	38

τ_2 acquires lock

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

Priority and Shared Resources

Task	Priority	Deadline	Arrival	Burst Length
τ_1	Highest	80	6	40
τ_2	Lowest	90	0	30

One CPU Quantum:
10 ticks

Tasks τ_1, τ_2 have a shared resource, will attempt to acquire it 5 ticks from their start, and relinquish it after 6 ticks after acquisition. Highest, medium, and lowest priority tasks will receive 3, 2, and 1 continuous CPU quantums, respectively.

0	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30	32	34	36	38
τ_2	τ_2	τ_2																	

τ_1 arrives, higher priority, schedule τ_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

Priority and Shared Resources

Task	Priority	Deadline	Arrival	Burst Length
τ_1	Highest	80	6	40
τ_2	Lowest	90	0	30

One CPU Quantum:

10 ticks

Tasks τ_1, τ_2 have a shared resource, will attempt to acquire it 5 ticks from their start, and relinquish it after 6 ticks after acquisition. Highest, medium, and lowest priority tasks will receive 3, 2, and 1 continuous CPU quantums, respectively.

0 τ_2	2	4	6 τ_1	8	10	12	14	16	18	20	22	24	26	28	30	32	34	36	38

quantum ends, τ_1 has highest priority, schedule τ_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

Priority and Shared Resources

Task	Priority	Deadline	Arrival	Burst Length
τ_1	Highest	80	6	40
τ_2	Lowest	90	0	30

One CPU Quantum:

10 ticks

Tasks τ_1, τ_2 have a shared resource, will attempt to acquire it 5 ticks from their start, and relinquish it after 6 ticks after acquisition. Highest, medium, and lowest priority tasks will receive 3, 2, and 1 continuous CPU quantums, respectively.

0 τ_2 2 4 | 6 τ_1 8 | 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38

τ_1 attempts to acquire lock, fails, is blocked, schedule τ_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

Priority and Shared Resources

Task	Priority	Deadline	Arrival	Burst Length
τ_1	Highest	80	6	40
τ_2	Lowest	90	0	30

One CPU Quantum:

10 ticks

Tasks τ_1, τ_2 have a shared resource, will attempt to acquire it 5 ticks from their start, and relinquish it after 6 ticks after acquisition. Highest, medium, and lowest priority tasks will receive 3, 2, and 1 continuous CPU quantums, respectively.

τ_2 (0–10)	τ_1 (10–20)	τ_2 (20–30)	16	18	20	22	24	26	28	30	32	34	36	38

τ_2 releases lock, τ_1 unblocks

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

Priority and Shared Resources

Task	Priority	Deadline	Arrival	Burst Length
τ_1	Highest	80	6	40
τ_2	Lowest	90	0	30

One CPU Quantum:

10 ticks

Tasks τ_1, τ_2 have a shared resource, will attempt to acquire it 5 ticks from their start, and relinquish it after 6 ticks after acquisition. Highest, medium, and lowest priority tasks will receive 3, 2, and 1 continuous CPU quantums, respectively.

0	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30	32	34	36	38
τ_2			τ_1			τ_2													

quantum ends, τ_1 scheduled, acquires lock

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

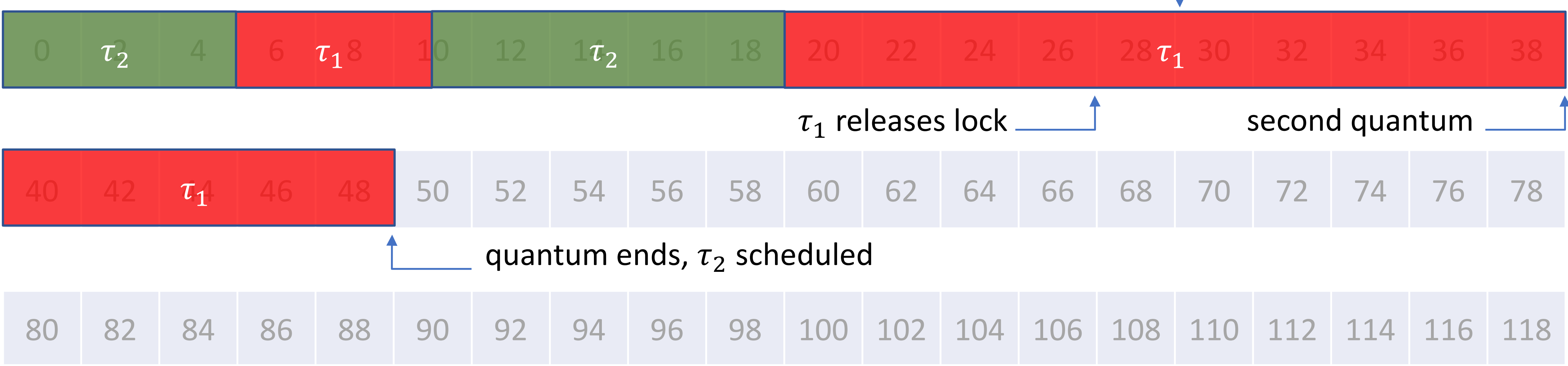
Priority and Shared Resources

Task	Priority	Deadline	Arrival	Burst Length
τ_1	Highest	80	6	40
τ_2	Lowest	90	0	30

One CPU Quantum:

10 ticks

Tasks τ_1, τ_2 have a shared resource, will attempt to acquire it 5 ticks from their start, and relinquish it after 6 ticks after acquisition. Highest, medium, and lowest priority tasks will receive 3, 2, and 1 continuous CPU quantums, respectively.



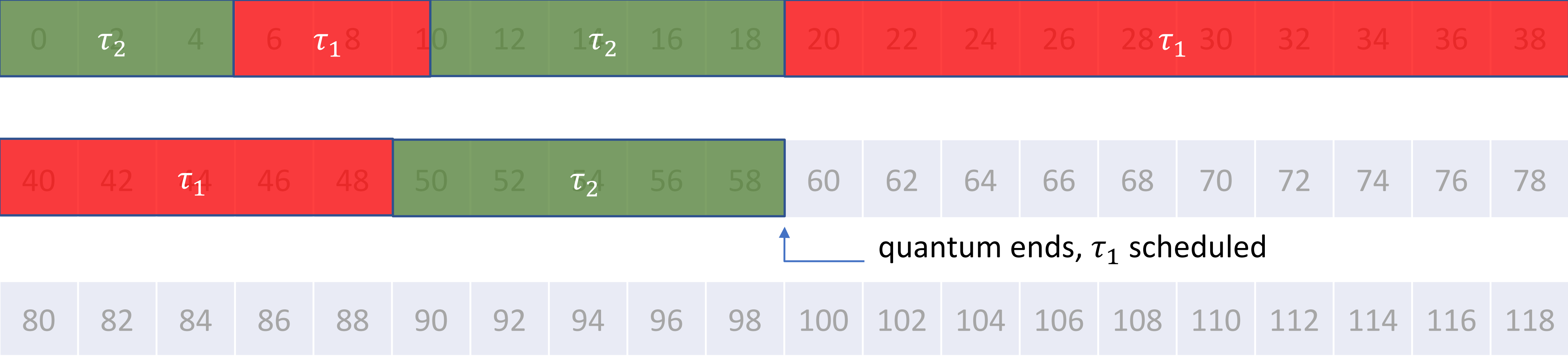
Priority and Shared Resources

Task	Priority	Deadline	Arrival	Burst Length
τ_1	Highest	80	6	40
τ_2	Lowest	90	0	30

One CPU Quantum:

10 ticks

Tasks τ_1, τ_2 have a shared resource, will attempt to acquire it 5 ticks from their start, and relinquish it after 6 ticks after acquisition. Highest, medium, and lowest priority tasks will receive 3, 2, and 1 continuous CPU quantums, respectively.



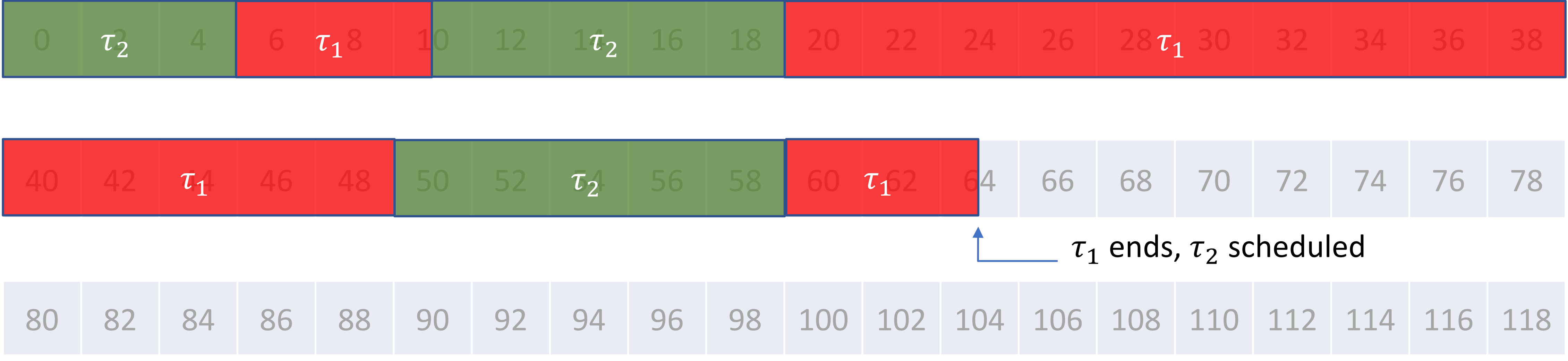
Priority and Shared Resources

Task	Priority	Deadline	Arrival	Burst Length
τ_1	Highest	80	6	40
τ_2	Lowest	90	0	30

One CPU Quantum:

10 ticks

Tasks τ_1, τ_2 have a shared resource, will attempt to acquire it 5 ticks from their start, and relinquish it after 6 ticks after acquisition. Highest, medium, and lowest priority tasks will receive 3, 2, and 1 continuous CPU quantums, respectively.



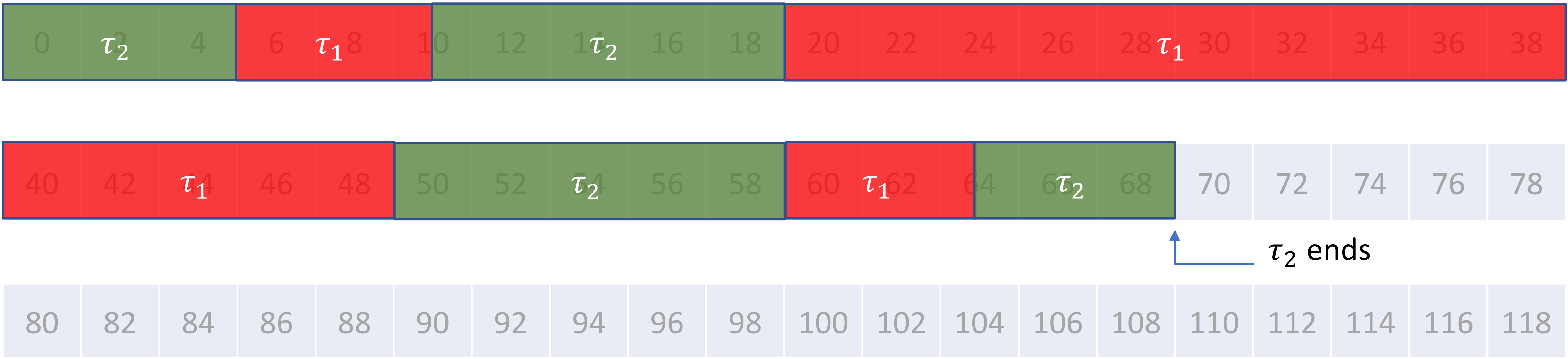
Priority and Shared Resources

Task	Priority	Deadline	Arrival	Burst Length
τ_1	Highest	80	6	40
τ_2	Lowest	90	0	30

One CPU Quantum:

10 ticks

Tasks τ_1, τ_2 have a shared resource, will attempt to acquire it 5 ticks from their start, and relinquish it after 6 ticks after acquisition. Highest, medium, and lowest priority tasks will receive 3, 2, and 1 continuous CPU quantums, respectively.



Priority and Shared Resources

Task	Priority	Deadline	Arrival	Burst Length
τ_1	Highest	80	6	40
τ_2	Lowest	90	0	30

One CPU Quantum:

10 ticks

Tasks τ_1, τ_2 have a shared resource, will attempt to acquire it 5 ticks from their start, and relinquish it after 6 ticks after acquisition. Highest, medium, and lowest priority tasks will receive 3, 2, and 1 continuous CPU quantums, respectively.

Priority and Shared Resources

Task	Priority	Deadline	Arrival	Burst Length
τ_1	Highest	80	6	20
τ_2	Lowest	90	0	30
τ_3	Medium	90	6	30

One CPU Quantum:

10 ticks

Same set of rules as before. τ_3 does not share any resources.

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

Priority and Shared Resources

Task	Priority	Deadline	Arrival	Burst Length
τ_1	Highest	80	6	20
τ_2	Lowest	90	0	30
τ_3	Medium	90	6	30

One CPU Quantum:

10 ticks

Same set of rules as before. τ_3 does not share any resources.

0	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30	32	34	36	38
τ_2																			

τ_2 acquires lock

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

Priority and Shared Resources

Task	Priority	Deadline	Arrival	Burst Length
τ_1	Highest	80	6	20
τ_2	Lowest	90	0	30
τ_3	Medium	90	6	30

One CPU Quantum:

10 ticks

Same set of rules as before. τ_3 does not share any resources.

0	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30	32	34	36	38
τ_2																			

τ_1 and τ_3 arrive, τ_1 has higher priority, schedule τ_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

Priority and Shared Resources

Task	Priority	Deadline	Arrival	Burst Length
τ_1	Highest	80	6	20
τ_2	Lowest	90	0	30
τ_3	Medium	90	6	30

One CPU Quantum:

10 ticks

Same set of rules as before. τ_3 does not share any resources.

0 τ_2	2	4	6 τ_1	8	10	12	14	16	18	20	22	24	26	28	30	32	34	36	38

quantum ends, τ_1 has highest priority, schedule τ_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

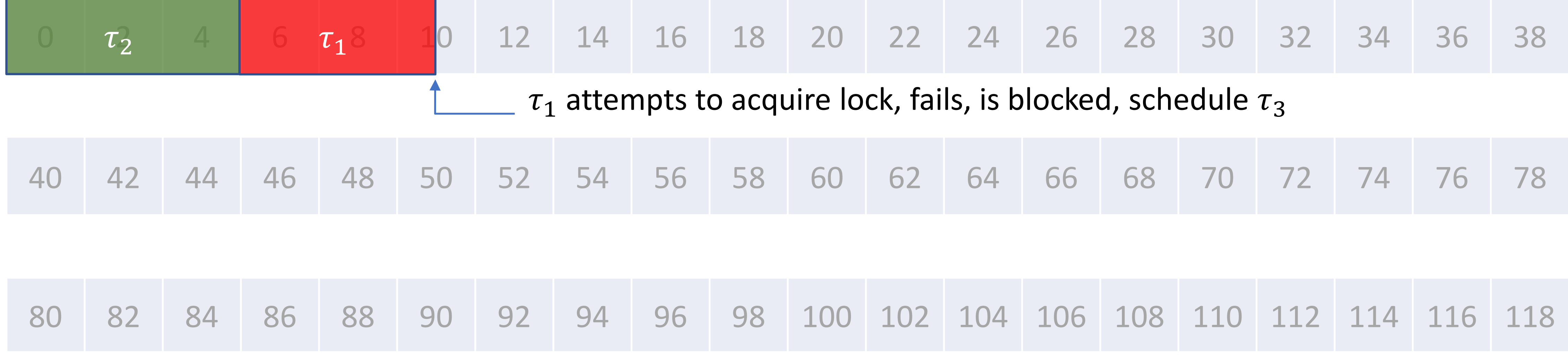
Priority and Shared Resources

Task	Priority	Deadline	Arrival	Burst Length
τ_1	Highest	80	6	20
τ_2	Lowest	90	0	30
τ_3	Medium	90	6	30

One CPU Quantum:

10 ticks

Same set of rules as before. τ_3 does not share any resources.



Priority and Shared Resources

Task	Priority	Deadline	Arrival	Burst Length
τ_1	Highest	80	6	20
τ_2	Lowest	90	0	30
τ_3	Medium	90	6	30

One CPU Quantum:

10 ticks

Same set of rules as before. τ_3 does not share any resources.

τ_2 (0, 2, 4)			τ_1 (6, 8, 10)			12	14	16	...	28	30	32	34	36	38

τ ... d, schedule τ_3

40	42	44	46	48	50	...	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

FAILURE!

Priority Inversion

Failure mode:

- Task τ_1 and τ_2 compete for a shared resource.
- τ_2 has lowest priority and τ_1 has highest priority.
- τ_2 acquires shared resource first.
- τ_3 shares no resources with either τ_1 or τ_2.
- τ_3 has higher priority than τ_2 but lower priority than τ_1

→ τ_3 executes with higher priority than τ_1, will not allow τ_2 to release shared resource!

Case Study – Mars Pathfinder

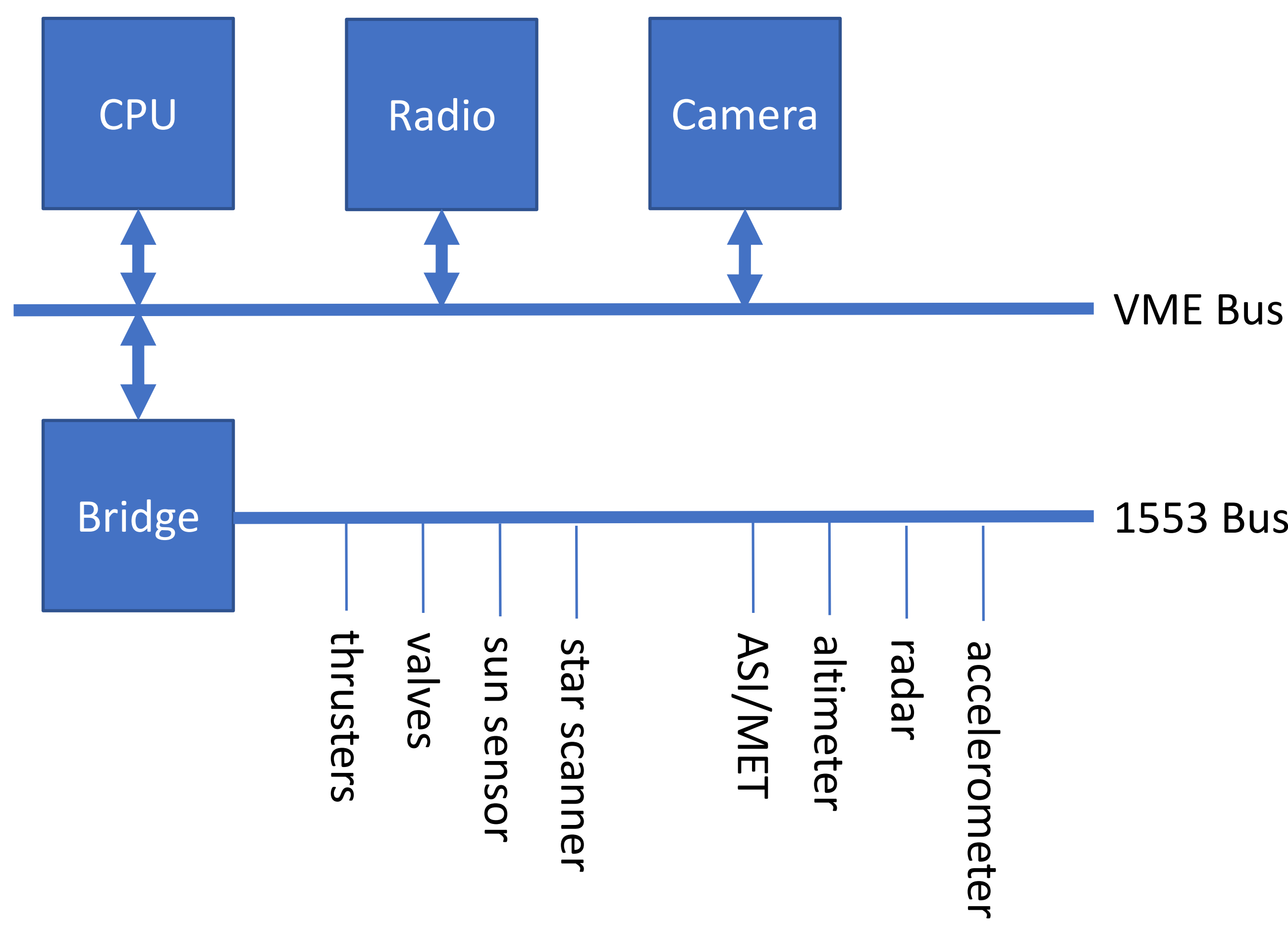
Launch: 4 December 1996

Operator: NASA/JPL

Lander: Pathfinder

- IBM Rad600 SC CPU
- 128 MB of RAM
- 6MB EEPROM
- VxWorks as OS

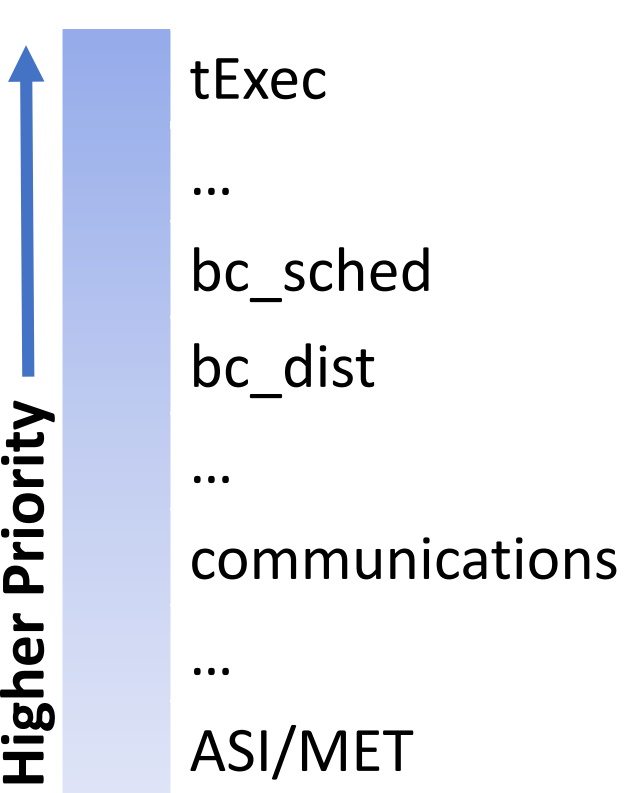
Case Study – Mars Pathfinder



Case Study – Mars Pathfinder

- Wind River's VxWorks as RTOS
- Preemtive, fixed priority scheduler
- RTOS employs a cyclic (round robin) scheduler at an 8 Hz rate
  - Entire task list executed in 0.125
- 1553 Bus management in two tasks
  - bc_sched
  - bc_dist
- Other tasks
  - Communications
  - ASI/MET
  - tExec (internal task in VxWorks)

Case Study – Mars Pathfinder



tExec

…

bc_sched

bc_dist

…

communications

…

ASI/MET

Case Study – Mars Pathfinder

- 1553 Bus Management
  - `bc_sched`: Transmission arbiter on the 1553 bus, transmits schedule for next bus cycle
  - `bc_dist`: Bus distribution task that decides who receives data
- Only one task can be employing the bus at a time
  - `bc_sched` checks whether `bc_dist` finishes
  - `bc_dist` uses `select()` to use VxWorks's `pipe()` facility to distribute data to ASI/MET task
  - ASI/MET task uses the `pipe()` facility to provide scientific data to transmit to radio (which scientists very much want here in Earth)
  - ASI/MET and `bc_dist` use a shared resource protected by a mutex

Case Study – Mars Pathfinder

- The Priority Inversion issue that went by unnoticed
  - ASI/MET calls `select()` and grabs mutex to update stored waitlist
  - ASI/MET is preempted by high priority task bc_dist
  - `bc_dist` can not acquire mutex lock and is blocked
  - ASI/MET task is preempted by medium priority tasks
  - `bc_sched` activates, sees `bc_dist` has not completed
  - `bc_dist` missed the deadline!

→ Programmed reaction to scheduler failure: Spacecraft resets, terminates all ground commands and reinitializes all hardware

Case Study – Mars Pathfinder

- Root issue: engineers did not anticipate the worst case scenario
  - Although arguably, this was a best case scenario
- Pathfinder's antenna worked better than expected
  - More data to send
  - Higher load on radio preempted the ASI/MET task for longer times
  - ASI/MET task had lock blocking `bc_dist`
  - Priority Inversion ensued

Question: How do we fix this issue?

Priority Inheritance

- After `bc_dist` blocks, ASI/MET task inherits the priority of bc_dist
- The ASI/MET task executes with the priority of `bc_dist` for as long as it holds the lock
- ASI/MET can no longer be preempted by communications task
- Upon releasing the lock, ASI/MET task returns to be a low priority task and is preempted by `bc_dist`
- System operates as normal

This is the fix that was sent to the Mars Pathfinder.

Priority Inheritance

In general: When a high priority task is blocked by a lower priority task due to a shared resource being held, the lower priority task is temporarily upgraded to high priority until its critical section finishes.

Speaking of the Antenna

- The task to transmit was only activated when transmission was possible
- This is not a periodic event
- This happens stochastically

→ This is an *aperiodic event*

Aperiodic Events

- Aperiodic events are events to which there are no limitations on arrival times.
- Follow the same semantics as other real-time tasks
  - Soft aperiodic events are those that execute without deadlines
  - Firm aperiodic events are those that execute with a deadline (worst case execution time must be known)

Dealing with Soft Aperiodic Events

- No deadline on execution
- Treat them as background tasks
- Schedule them when there is nothing else to do
- Queue them in a round robin fashion
  - Keep two execution queues: one for real-time tasks and one for soft aperiodic events
  - Give priority to real-time tasks
  - Any issues with this?
  - Relies on CPU time being available for the background task queue (CPU starvation)

Dealing with Soft Aperiodic Events

- Better solution: use a *task server*
- The *task server* is a periodic event whose only job is to service aperiodic events
  - Almost like a scheduler for background tasks that gets scheduled every once in a while
- Being a real-time task, task servers have
  - A period T
  - A burst length C (called the capacity of the server)
- At every period T, the server becomes active and services any aperiodic event using its capacity C
- If no aperiodic events are waiting to be serviced, the task yields.

Task Servers

Given:

- Aperiodic event with burst length C_a and hard deadline D_a
- Task server with capacity C_s and period T_s

The worst case deadline of the single task has the relation

$$T_s + \left\lceil \frac{C_a}{C_s} \right\rceil \times T_s \leq D_a$$

Task Servers

- Two types of priority
  - *Fixed priority*: based on Rate Monotonic Scheduling
  - *Dynamic priority*: based on Deadline Monotonic Scheduling
- Need to measure how aperiodic events will be serviced by the server
- Need to measure the effect of the server in periodic tasks
  - Remember the Mars Pathfinder: communications woke up too often

Polling Server

- Fixed priority server executed as regular task
- Aperiodic events are served only during the execution interval of server
- Implementation:
  - Server has a run queue for aperiodic events
  - Control capacity used by aperiodic events
- Response time:
  - Better than background execution
  - Aperiodic events are guaranteed CPU time by the server
- Impact on periodic task set:
  - Same as that of a periodic task
  - Server itself is a periodic task

Deferrable Server

- Fixed priority server, allowed to execute at any instant
- Handles aperiodic events until either *the end of its period* or *its capacity gets exhausted*
- Capacity is usually equal to its burst length [but this is *not* mandatory] and gets replenished at the start of each period
- Response time:
  - Improved over a polling server
- Impact on periodic task set:
  - Negative impact on schedulability of periodic tasks
  - Delayed execution increases load on the future
  - Possible that server executes back to back, causing other periodic tasks to miss deadlines

Sporadic Server

- Fixed priority server similar to deferrable server
- We do not allow the sporadic server to penalize the schedulability of periodic tasks
  - Do not replenish its capacity at the start of every period
  - Only replenish capacity one period after it is used
- Response time:
  - Similar to deferrable servers
- Impact on periodic tasks
  - The same as a polling server

Total Bandwidth Server

- Dynamic priority server
- Executes aperiodic events as soon as possible while preserving bandwidth it has
- When the ith periodic event arrives at time $t = a_i$, it receives a deadline given by

$$d_i = \max(a_i, d_i) + \frac{C_i}{U_s}$$

  where, by definition $d_0 = 0$, C_i is the event burst length, and U_s is the server utilization (bandwidth)
- Once deadline is computed, server is inserted into ready queue and is treated as any other periodic event

Total Bandwidth Server

- Server requires a priori knowledge of the execution time of the event
- If event executes longer than declared, server may overrun scheduler
- Designed to work with Deadline Monotonic Scheduler
- Given set of n periodic events with processor utilization U_p, and a total bandwidth server with utilization U_s, the whole set is schedulable using DMS if and only if

$$U_p + U_s \leq 1$$

Constant Bandwidth Server

- Dynamic priority server
- Developed to solve the robustness issues that arise with Total Bandwidth Servers
- Enforces bandwidth isolation by managing execution time and capacity
- Aperiodic event τ_i is reserved a *maximum runtime* C_s every *reservation period* T_s

Constant Bandwidth Server

- When the event first arrives, it is given a scheduling deadline of $d_i^s = 0$ and a current runtime $q_i = 0$
- We check if the current scheduling deadline can be used

$$d_i^s > t \wedge \frac{q_i}{d_i^s - t} < \frac{C_s}{T_s}$$

- If not, set

$$d_i^s = t + T_s$$
$$q_i = C_s$$

- When τ_i executes for a time δ,update

$$q_i = q_i - \delta$$

Constant Bandwidth Server

- If q_i reaches 0, then τ_i can not be scheduled until time d_i^s, at which point we update the values

$$d_i^s = d_i^s + T_s$$
$$q_i = q_i + C_s$$

- Keep updating deadline of aperiodic event until it finishes completion

→ Aperiodic task can no longer overrun the scheduler

Creating Aperiodic Events

- Done in response to a stimulus/input
- The RTOS must allocate resources for the new task
  - Stack regions
  - Data regions
  - Code regions
- The RTOS must add the newly created task to the list of runnable tasks
  - If using a server, the server must be notified of the new task
- Task is executed as necessary

Creating Aperiodic Events

Example assumptions:

- Tasks creation happens when the CPU is in Handler Mode
- Tasks are executed in non-privileged mode
- Tasks utilize their own stack
- Code for task has been loaded already
- Task will be run as a background service

Creating Aperiodic Events

```
/* void* init_user_stack(void* stack_address, void* entry_point) */
init_user_stack:
        mov r3, #(1 << 24)          /* xPSR with Thumb bit set */
        mov r2, r1                  /* handler return address is entry point */
        mov r1, #-1                 /* lr to top of memory */
        stmdb r0!, {r1, r2, r3}     /* store top of frame */
        mov r1, #13                 /* prepare to initialize other registers */
        eor r2, r2, r2              /* clear r2 */
1:      subs r1, r1, #1             /* push 0 into stack for all registers */
        stmdb r0!, {r2}             /* that need to be initialized */
        bne 1b                      /* keep looping until all registers */
        bx lr                       /* return */
```

When CPU leaves handler mode, it will load the stack frame from psp. The CPU will leave handler mode and start executing the newly created task.

Exiting Aperiodic Events

- Aperiodic events will eventually terminate
- RTOS must deallocate resources related to the task
- RTOS must remove the event from the task queue
- If necessary, RTOS must destroy all references to locks held by the task
- RTOS may need to keep the Process Control Block for the task if its termination must be handled by another task

→ Task must notify OS it wishes to exit

→ `sys_exit`

Exiting Aperiodic Events

```
/* C library function */
extern void __exit(int retcode);
void exit(int retcode) {
        __exit(retcode);
}

/* assembly wrapper */
__exit:
        svc #6
1:      br 1b
```

Exiting Aperiodic Events

```
/* kernel interface */
void sys_exit(int retcode) {
        deallocate_resources(current);
        destroy_locks(current);
        if(current->has_parent) {
                zombify(current);
                send_signal(current->ppid, SIGCHLD, current->pid, retcode);
        }
}
```