Background: OS and RTOS

EEL 4745C: Microprocessor Applications II

Fall 2022

Md Jahidul Islam

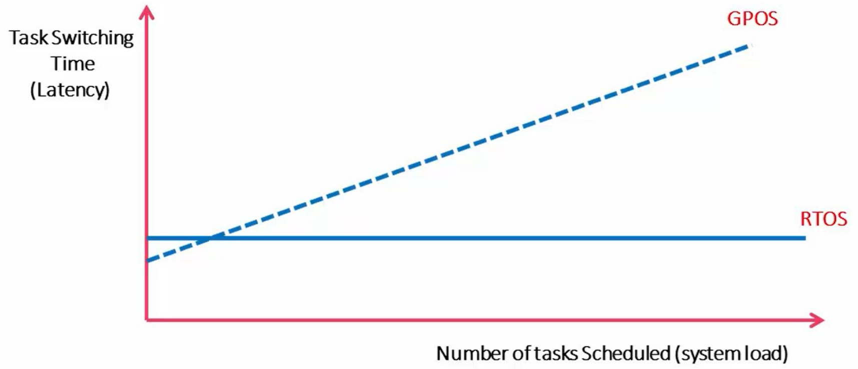
Lecture 4

Department of Electrical & Computer Engineering
UNIVERSITY of FLORIDA

Topics and Outline

OS Concepts and RTOS adaptations

- Programs and processes
- Threads in multi-threaded systems
- Scheduling algorithms and implementation
- Inter-process communication
- Synchronization and resource sharing



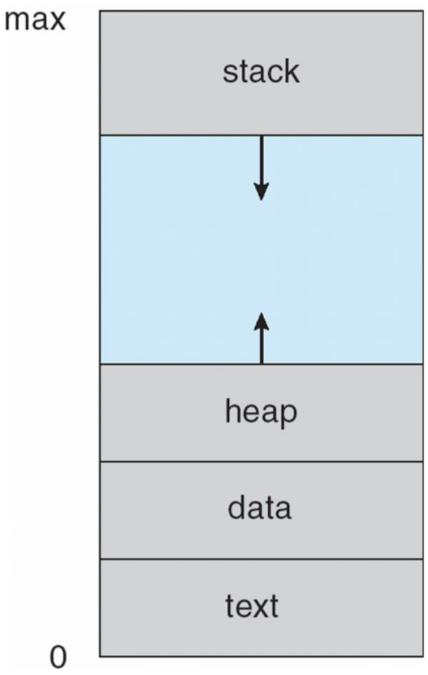
Reference and acknowledgements

- **Book:** *Operating System Concepts* (Ninth Edition) By A. Silberschatz, P. Galvin, and G. Gagne
- **Course:**
  - *Operating Systems* By Dr. Steven Hand at University of Cambridge
  - *An introduction to RTOS and Schedulability Analysis* By Marco Di Natale Scuola Superiore S. Anna

Processes (aka Jobs)

- An operating system executes a variety of programs:
  - Batch system – **jobs / processes**
  - Time-shared systems – **user programs** or **tasks**

- **Process** – a program in execution
- Multiple parts
  - The program code, also called **text section**
  - Current activity: **PC (program counter)**, processor registers
  - **Stack** containing temporary data
    - Function parameters, return addresses, local variables
  - **Data section** containing global variables
  - **Heap** containing memory dynamically allocated during run time

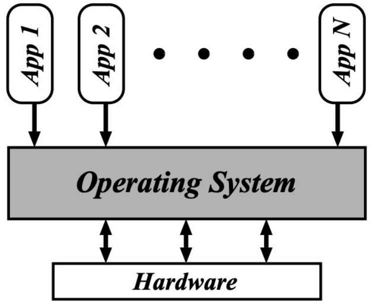


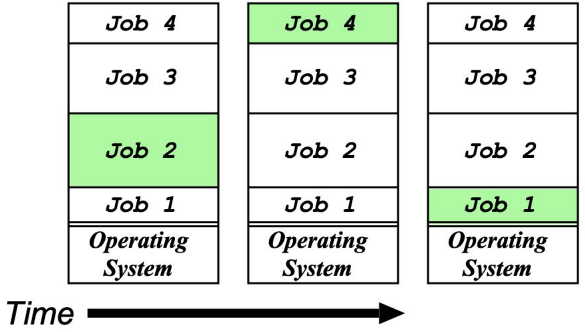
Program vs Process vs Threads

- **Program** is a ***passive*** entity stored on disk (executable files)
- **Process** is ***active***

- Program becomes process when it is loaded into memory
- Execution of program started via
  - GUI or mouse clicks, command line calls, etc.
  - Interrupts or calls by other programs!

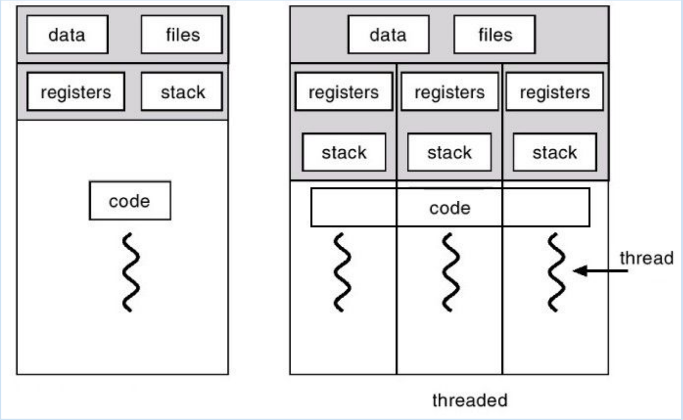
- One program can be several processes
- Each process can have multiple **threads**
  - **A thread** is the basic unit to which OS allocates processor time
  - Each process is started with a primary thread
  - But can create additional threads from any of its threads.



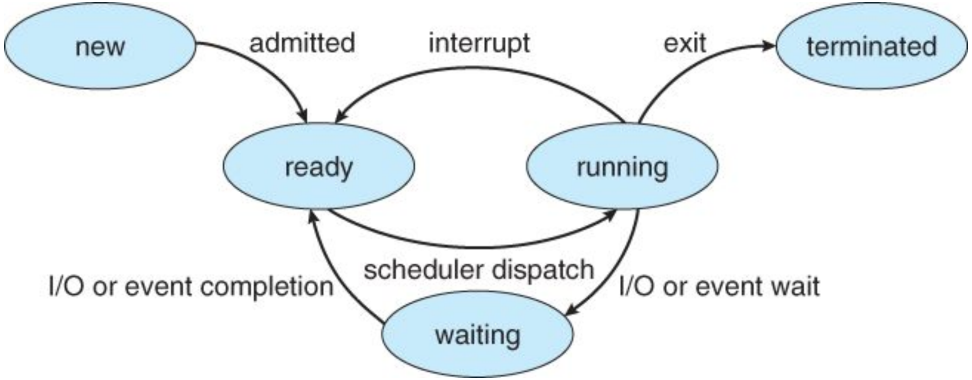


Process States

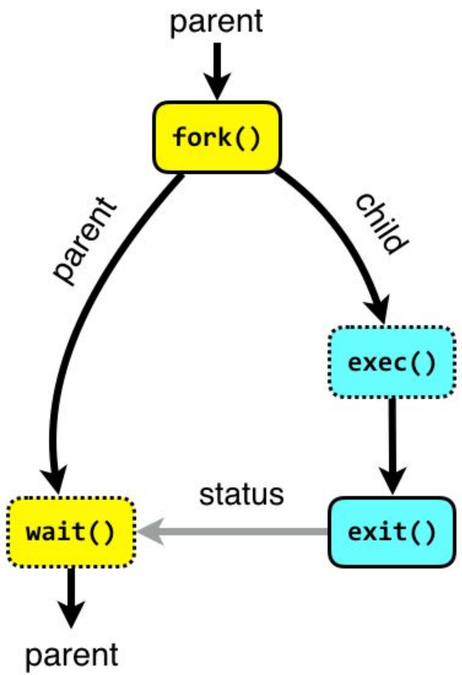
- As a process executes, it changes **state**
- OS is responsible for coordination
  - Multi-threaded scheduling & execution



- Process states
  - **new**: The process is being created
  - **running**: Instructions are being executed
  - **waiting**: The process is waiting for some event to occur
  - **ready**: The process is waiting to be assigned to a processor
  - **terminated**: The process has finished execution



Process Creation



A parent process can create many new processes via system calls

- System call to create process: `fork()`
- Each child process may in-turn create new child process
- Every process gets a unique process identifier: PID

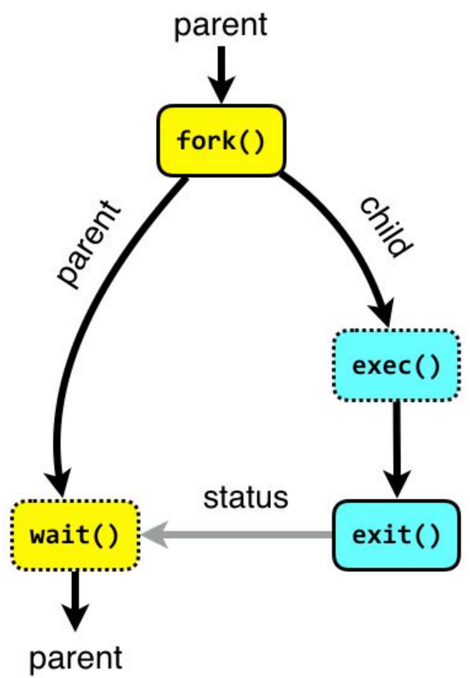
How the child process gets its resources?

- OS can create some
- The parent process can allocate some

Address space of child processes

- Gets an exact copy of the parents address space
- *What is 'copy-on-write'?*

Process Termination



A process termination can occur in many ways

- Normal termination – A process finishes executing its final statement: `exit()`. All the resources allocated to it are freed by the operating system.
- Forced Termination – a parent process can terminate its child process by invoking the system call: `abort()`.
- This can happen due to the following reasons:
  - Child exceeds its usage of resources
  - Task assigned to the child is no longer required
  - Parent exits; OS does not allow child to run if parent terminates, child is then handled by the init_process.
  - User can also forcefully terminate a process: `kill()`
- If no parent waiting (didn't invoke `wait()`), process is: **zombie**
- If parent terminated without invoking wait , process is: **orphan**

Logistics

⇒ Lab-2 demo and quiz-1 starts today

- Different problems, but similar difficulty level
- One of three (3 x 5 = 15)

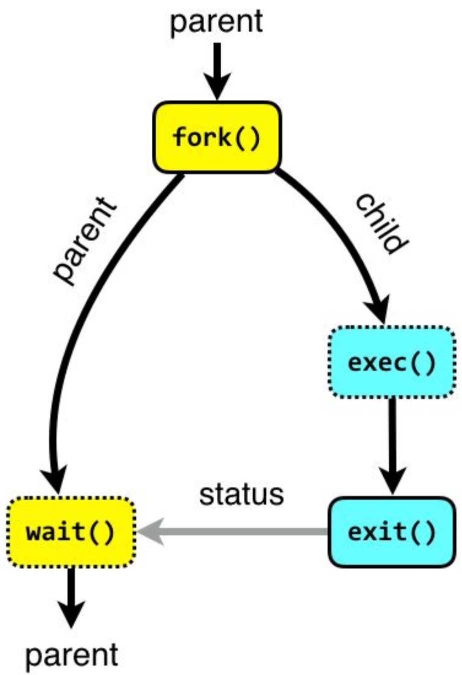
⇒ Lab-1 grades are out (since Monday)

- Grades are final after 1 week of posting

⇒ Lab-3 manual and code template are out

- *Go over the files and functions (specially the new IPC library and periodic thread functions)*
- *Read the manual carefully and thoroughly*

Zombie vs Orphan Process



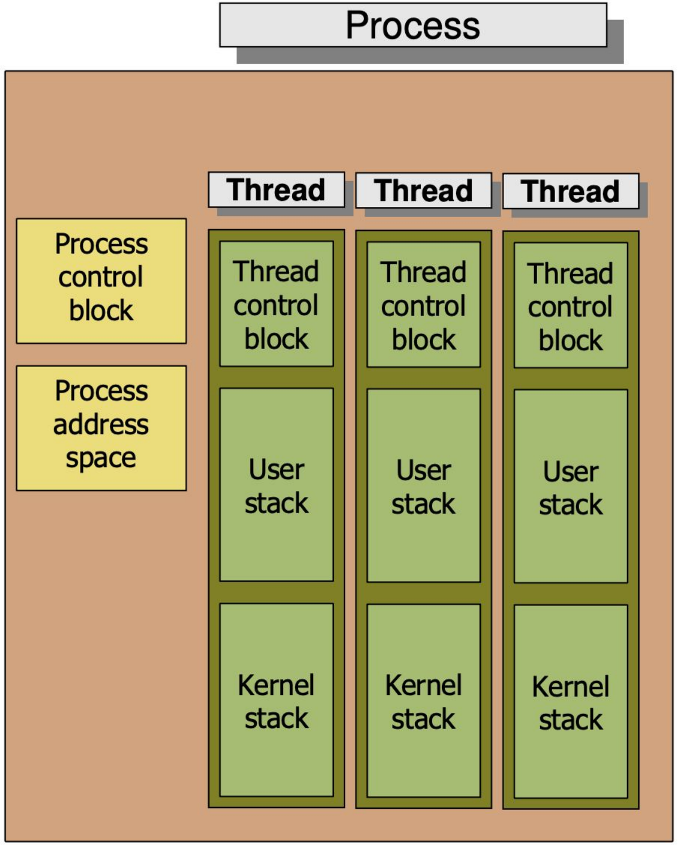
- When a child process terminates before the parent invokes `wait()`
  - It needs to signal its parents about its exit: using `SIGCHLD`
  - Then the parent calls `wait()` and clears it from process table
  - During this step the child process is a ***zombie or defunct***

    A process that has completed its task while no parent is waiting on it, but still shows an entry in the process table

- When a parent is terminated but the child process is still running
  - It is called an **orphan**
  - Orphan processes are handled by the *init_process*, which performs the `wait()` call so that the orphans processes can *die*

>> See this stackoverflow discussion.

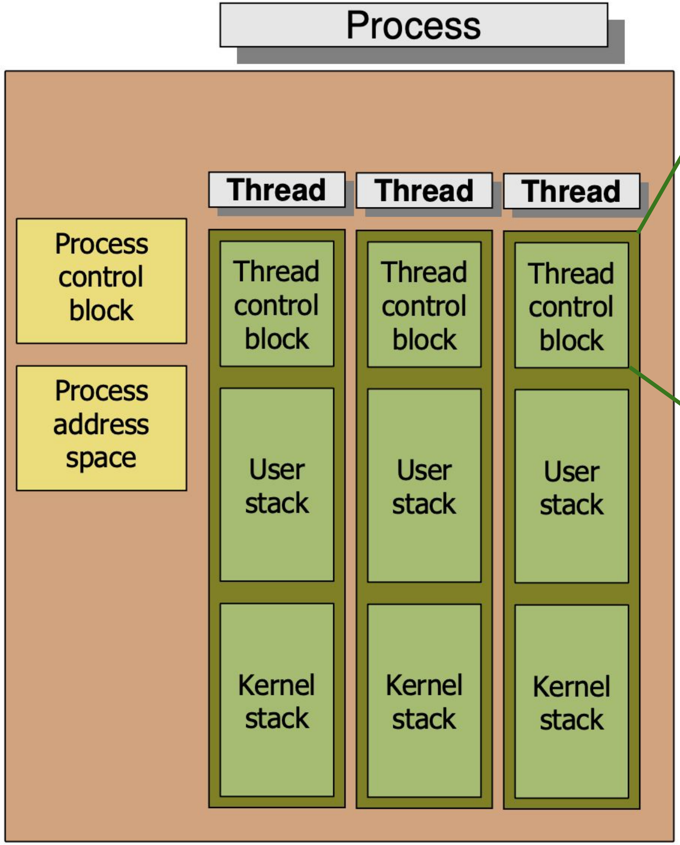
PCB: Process Control Block

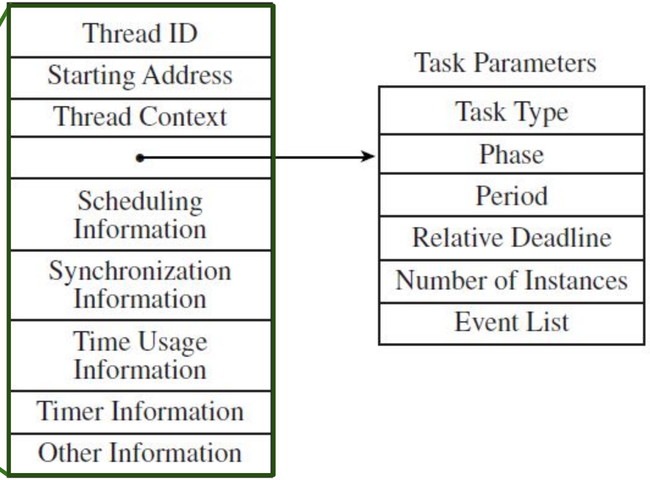


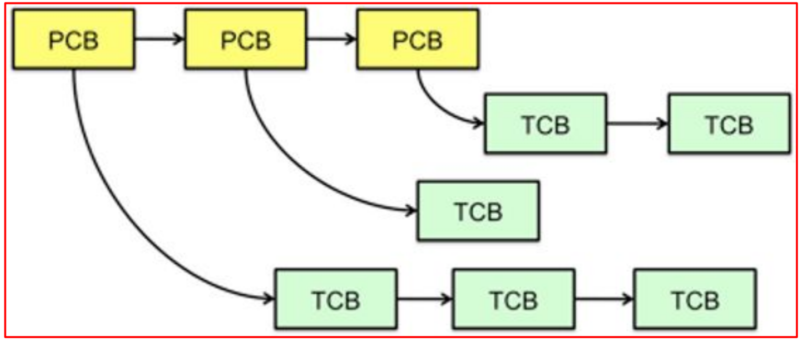
PCB Holds information associated with each process

- Process state
- Program counter
- CPU registers
- CPU scheduling information
  - Priorities, scheduling queue pointers
- Memory-management information
  - Memory allocated to the process
- Accounting information
  - CPU used, clock time elapsed since start, time limits
- I/O status information
  - I/O devices allocated to process, list of opened files

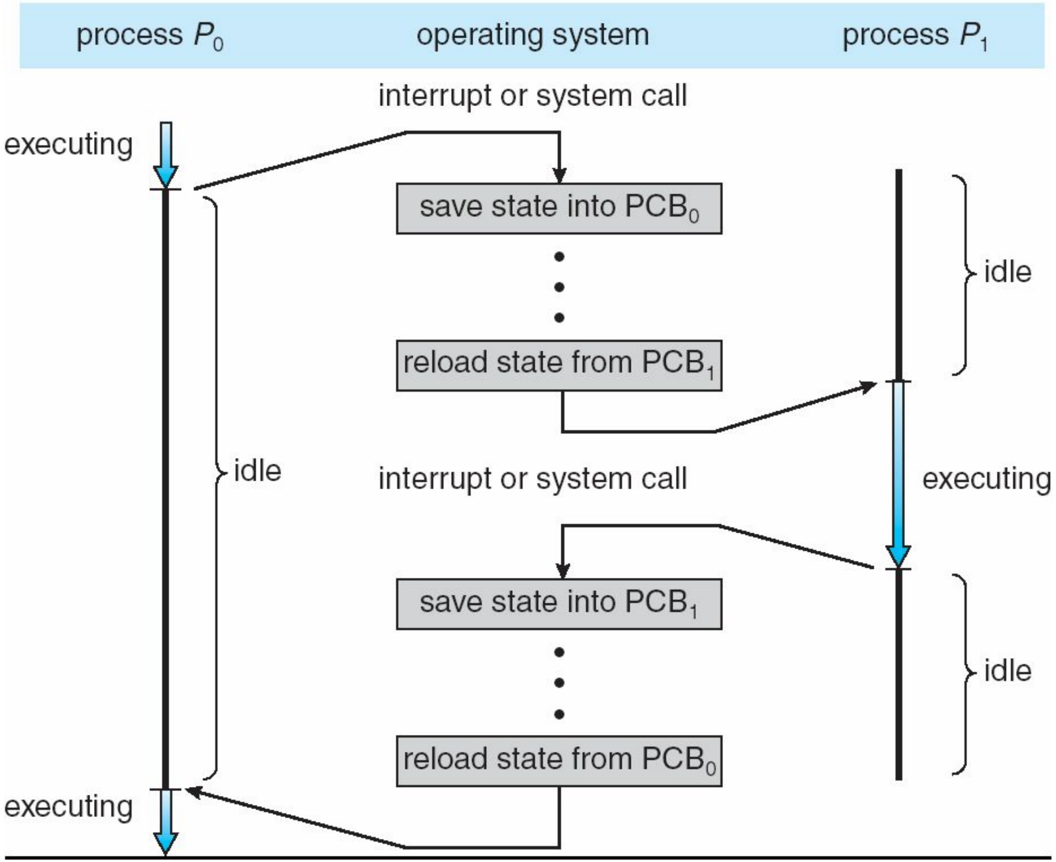
Control Blocks: PCB and TCB



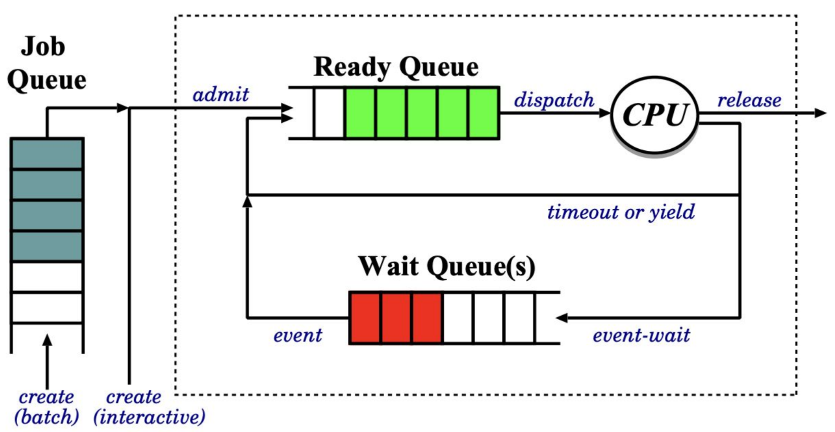




Process-to-Process Transition



Process Scheduling

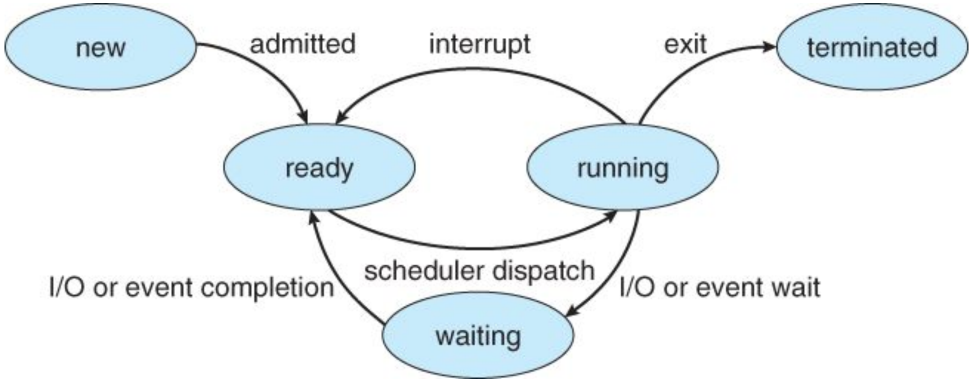


- Maximizes CPU use for time sharing
- Maintains **scheduling queues** of processes
  - Job queue – set of all processes in the system
  - Ready queue – set of all processes residing in main memory, ready and waiting to execute
  - Device queues – set of processes waiting for an I/O device
  - Processes migrate among the various queues

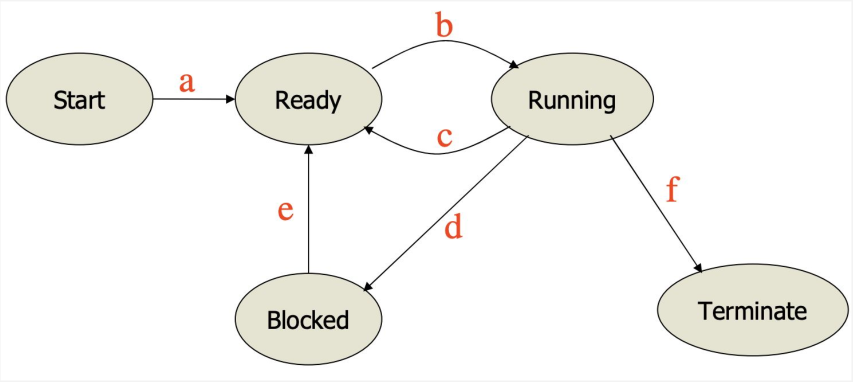
Scheduling: Process vs Thread

- **Scheduling a process** means making the threads within the process candidates for scheduling
- **Scheduling a thread** means resuming it

- Suspending a process means suspending all the threads within the process.
- Suspending a thread means suspending its execution

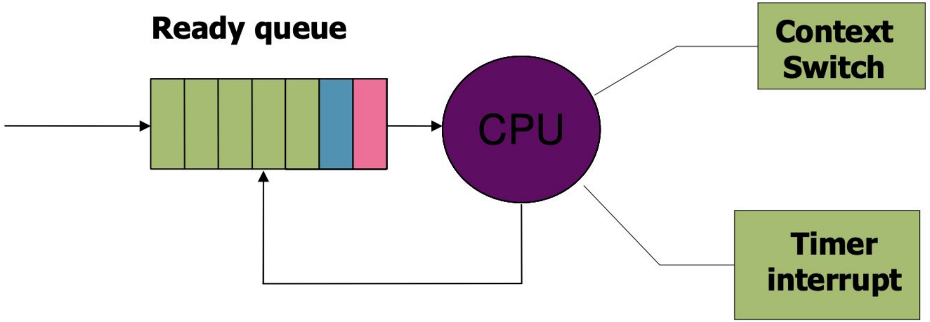


Process States



Thread States

Context Switching: PCB and TCB



- When CPU switches to another process
  - OS must **save the state** of the old process and load the **saved state** for the new process
- **Context** of a process represented in the PCB
- **Context-switch** time is overhead; the system does no useful work while switching

- Context switching between process and threads uses the same philosophy
  - Thread context switching (saving and loading new TCBs) are obvious much faster

IPC: Inter-Process Communication

- Processes within a system may be ***independent*** or ***cooperating***
- Cooperating process can affect or be affected by other processes, including sharing data
- Reasons for cooperating processes:
  - Information sharing
  - Computation speedup, Modularity
- Cooperating processes need **interprocess communication** (**IPC**)
- Two models of IPC
  - **Shared memory**
    - Need synchronization
  - **Message passing**
    - `send (P, message)` – send a message to process P
    - `receive(Q, message)` – receive a message from process Q

Synchronization: Message Passing

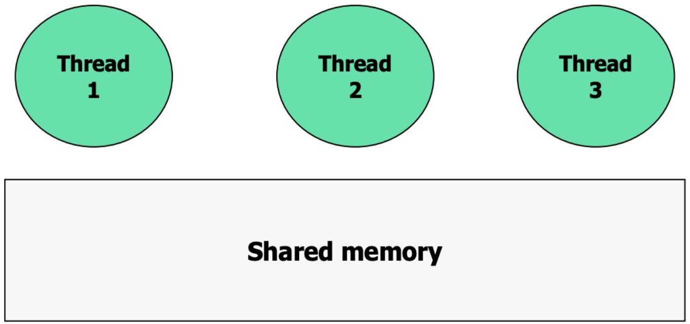
- Message passing may be either blocking or non-blocking
- **Blocking** is considered **synchronous**
  - **Blocking send** -- the sender is blocked until the message is received
  - **Blocking receive** -- the receiver is blocked until a message is available

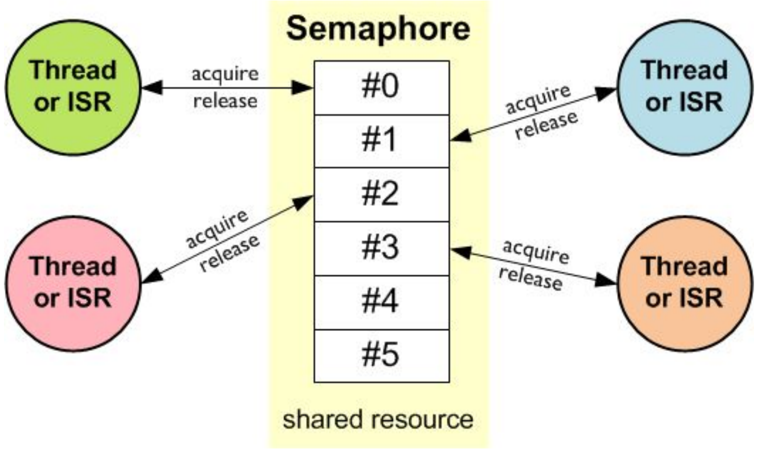
- **Non-blocking** is considered **asynchronous**
  - **Non-blocking send** -- the sender sends the message and continue
  - **Non-blocking receive** -- the receiver receives:
    - A valid message, or
    - Null message

- Different combinations possible
  - If both send and receive are blocking, we have a **rendezvous**

Synchronization: Resource Sharing

- **Shared memory** communication
  - Simplest model and the closest to the machine
  - all threads can access the same memory locations
- **Critical Section**
  - Parts of the code where the problem may happen
  - A sequence of operations that cannot be interleaved
- Resource: shared object where the conflict may happen
- Two critical sections on the same resource must be properly sequentialized, ie, must execute in mutual exclusion
  - General solution: **semaphores**!

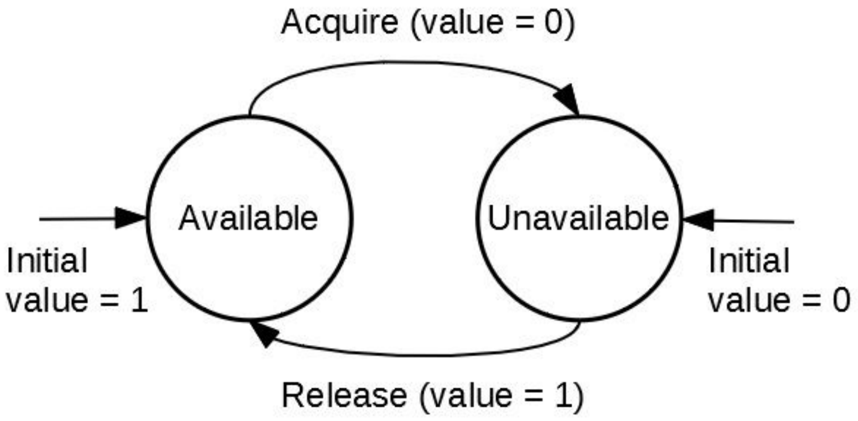




Synchronization: Semaphores

⇒ Software-based thread synchronization

- Synchronization with just a shared integer ~ **semaphore**
- Proposed by Edsger Dijkstra
- Types:
  - Counting semaphores (when ***N*** units of resources available)
  - Binary semaphores (guarantees mutual exclusiveness)



P

```
wait (S) {
    while S <= 0
        ; // no-op
    S--;
}
```

V

```
signal (S) {
    S++;
}
```

Semaphore: General Implementation

```
typedef struct {
        <blocked queue> blocked;
        int counter;
} sem_t;

void sem_init       (sem_t &s, int n);

void sem_wait       (sem_t &s);
void sem_post       (sem_t &s);
```

```
sem_t s;
...
sem_init(&s, 1);
```

```
void *threadA(void *arg)
{
        ...
        sem_wait(&s);
          <critical section>
        sem_post(&s);
        ...
}
```

```
void *threadB(void *arg)
{
        ...
        sem_wait(&s);
          <critical section>
        sem_post(&s);
        ...
}
```

```
void sem_init (sem_t *s, int n)
{
  s->count=n;
  ...
}
```

```
void sem_wait(sem_t *s)
{
  if (counter == 0)
                <block the thread>
  else
                counter--;
}
```

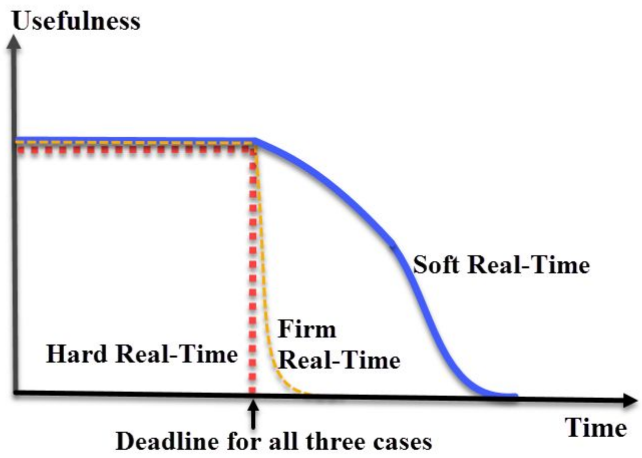
```
void sem_post(sem_t *s)
{
  if (<there are blocked threads>)
                <unblock a thread>
  else
                counter++;
}
```

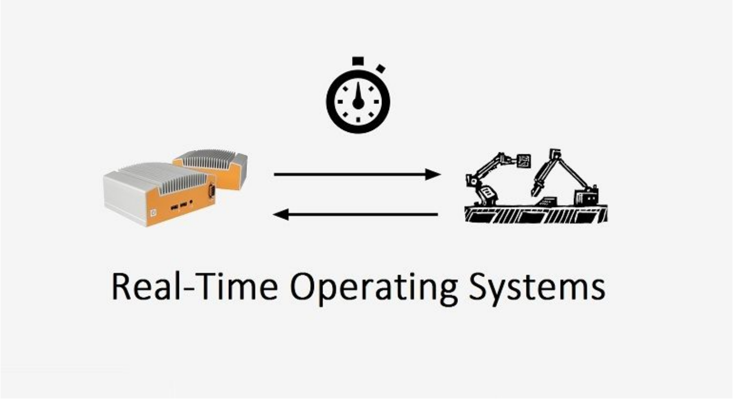
RTOS Adaptation

Scheduling and Synchronization

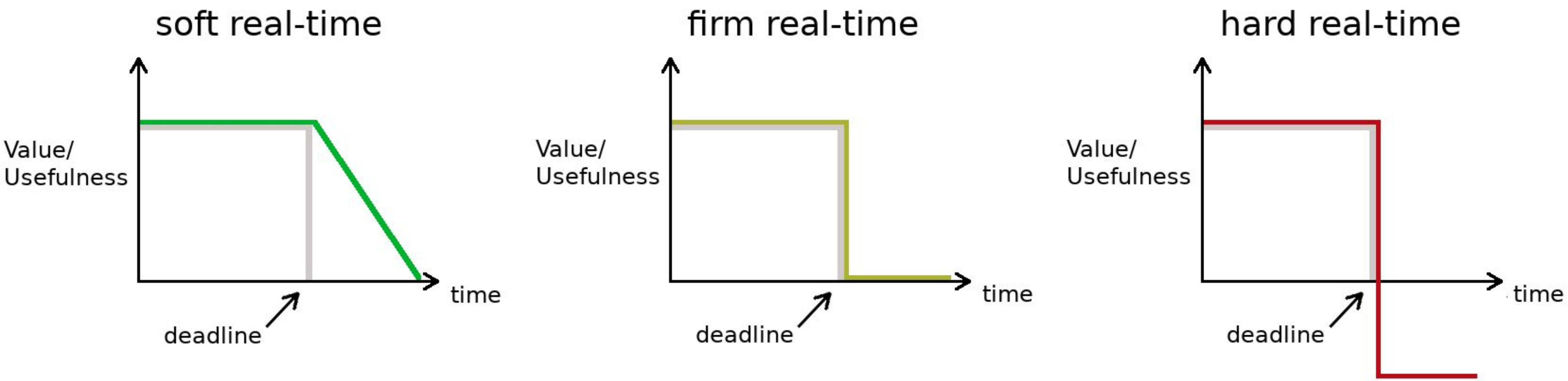
RTS: Real-time Systems

- **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
- **For timing guarantee, system must be predictable**
  - Upper bound suffices for most cases





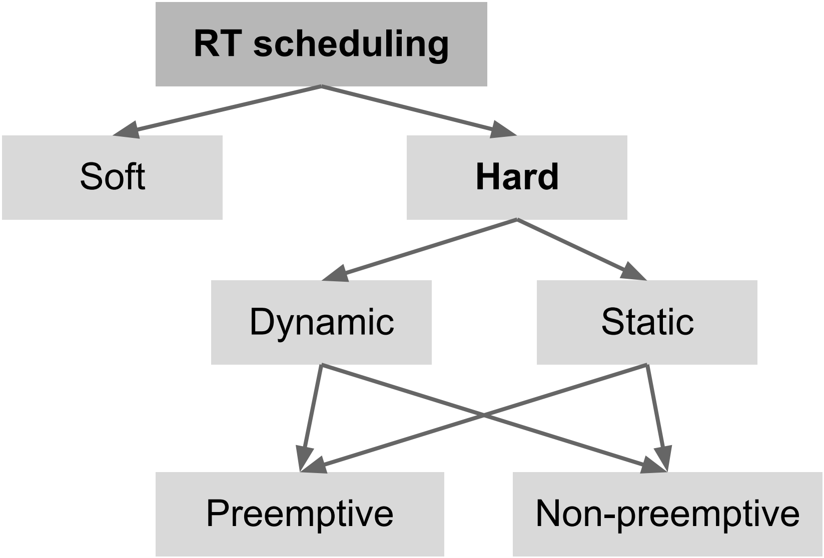
RTS Deadlines: Soft vs Firm vs Hard



- **Examples**
  - Hard deadline: traffic controllers
  - Soft/Firm deadline: background downloads, games, and multimedia

- **Various types of tasks in RTS/RTOS**
  - Time: periodic vs aperiodic vs sporadic
  - Interrupt: preemptive vs non-preemptive
  - Priority/compile-time: static vs dynamic

Scheduling Algorithms



> We will explore:

- **Soft scheduling**
  - RR: Round-robin scheduling
- **Hard (real-time) scheduling**
  - Rate Monotonic scheduler
  - Deadline Monotonic scheduler

> Other important/famous algorithms: **see here**

Round-robin Scheduling

⇒ Each job gets equal CPU time - no priority:

- Circular queue
- Fair but inefficient

Given:

- Circular list of tasks `task_list` of size `N`
- Number of scheduling ticks: `t`

schedule:

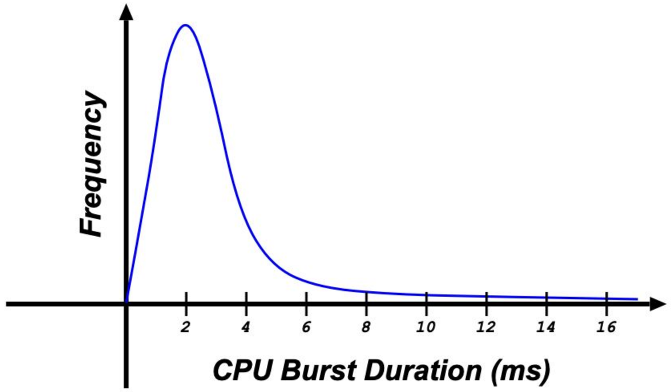
- `current =  task_list[t]`
- `t = (t + 1) mod N`

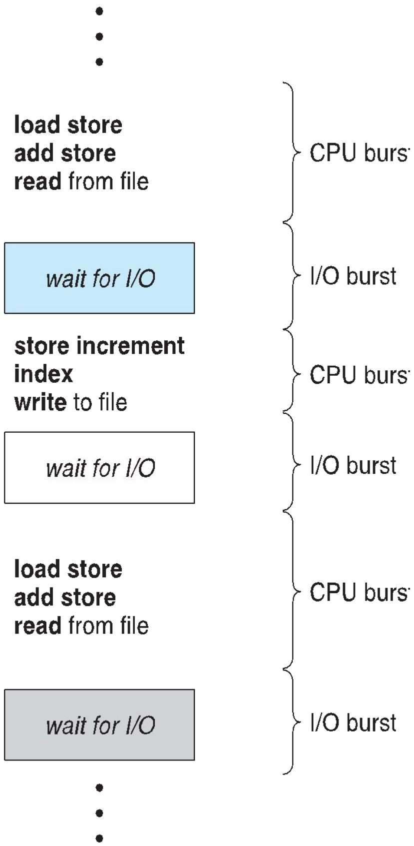
Task	Period	Arrival	Burst Length
τ_1	80	0	30
τ_2	40	8	10
τ_3	100	0	15

- *If the one CPU quantum is 5 ticks, can you track the time-process horizon? (aka Gantt chart)*
- *Comment of CPU Efficiency!*

Digressing: CPU–I/O Burst Cycles

- 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**





Back: 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

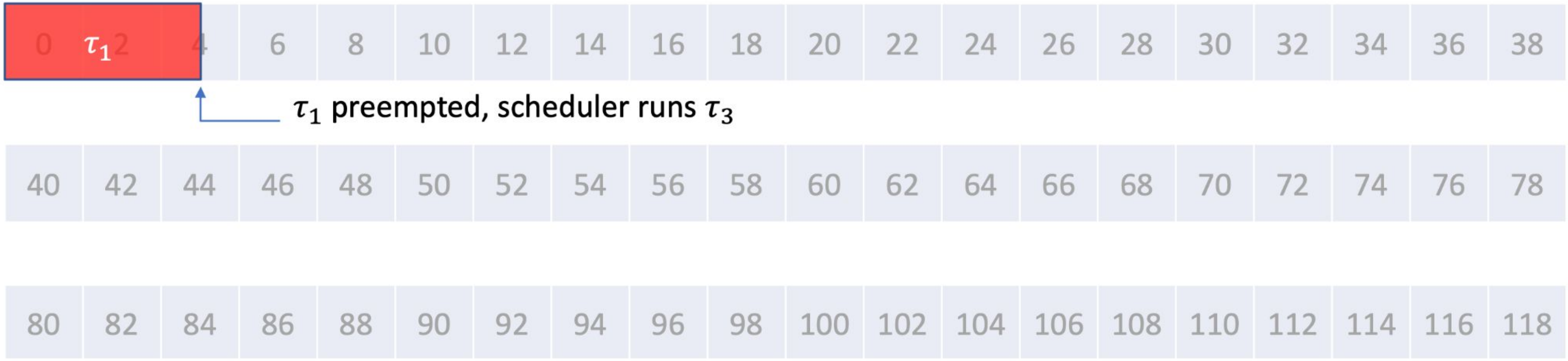
Round-robin Scheduling (Contd.)

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: $\{\tau_1, \tau_3\}$

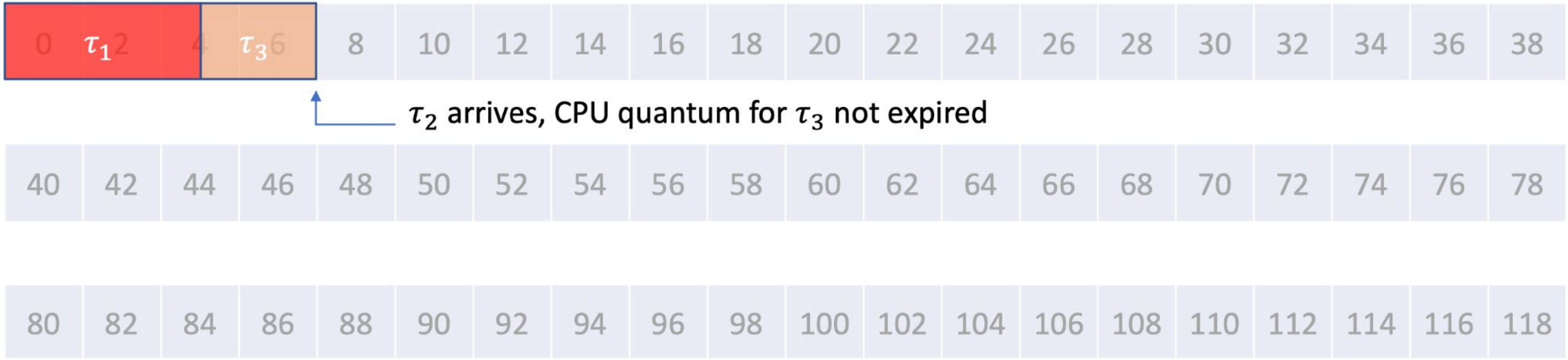


Round-robin Scheduling (Contd.)

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: $\{\tau_1, \tau_3, \tau_2\}$

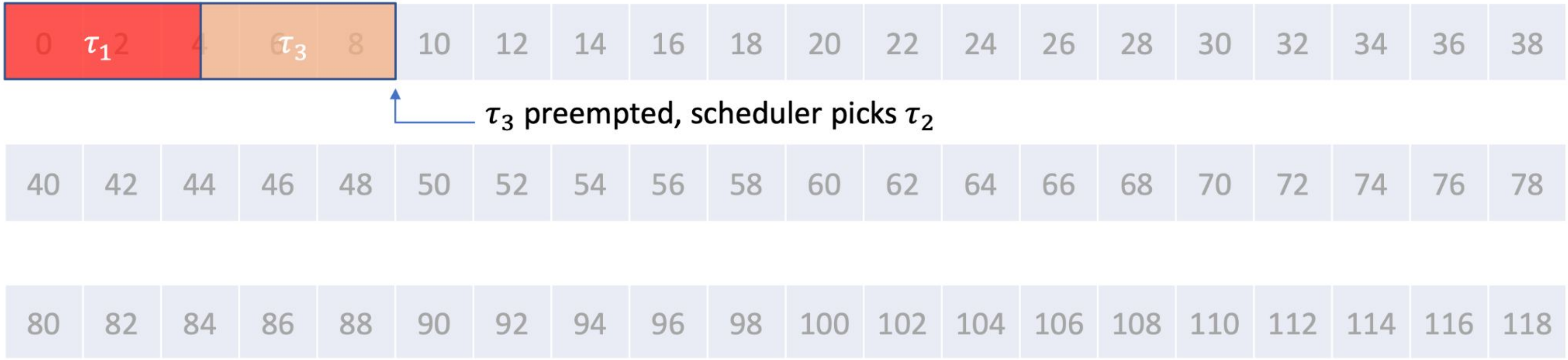


Round-robin Scheduling (Contd.)

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: $\{\tau_1, \tau_3, \tau_2\}$



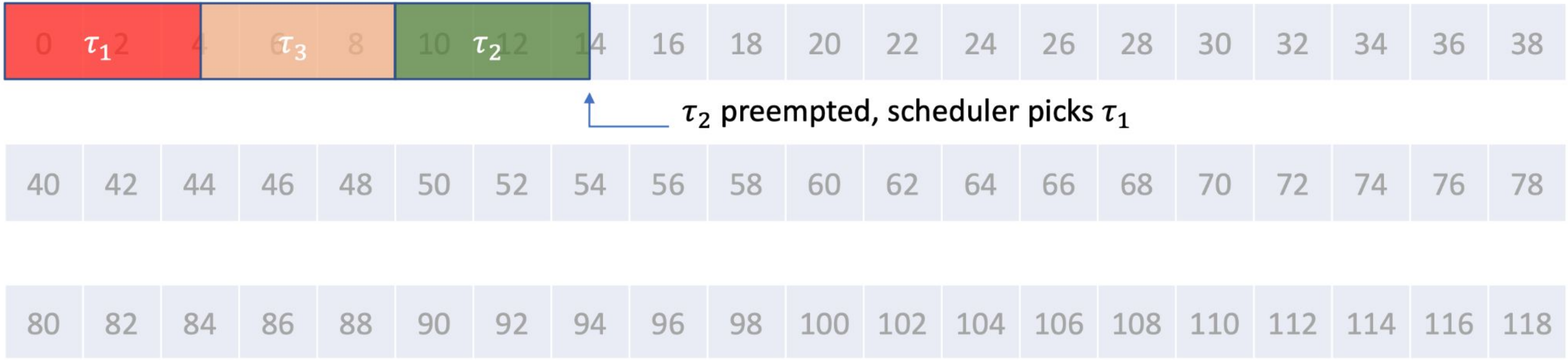
Round-robin Scheduling (Contd.)

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: $\{\tau_1, \tau_3, \tau_2\}$

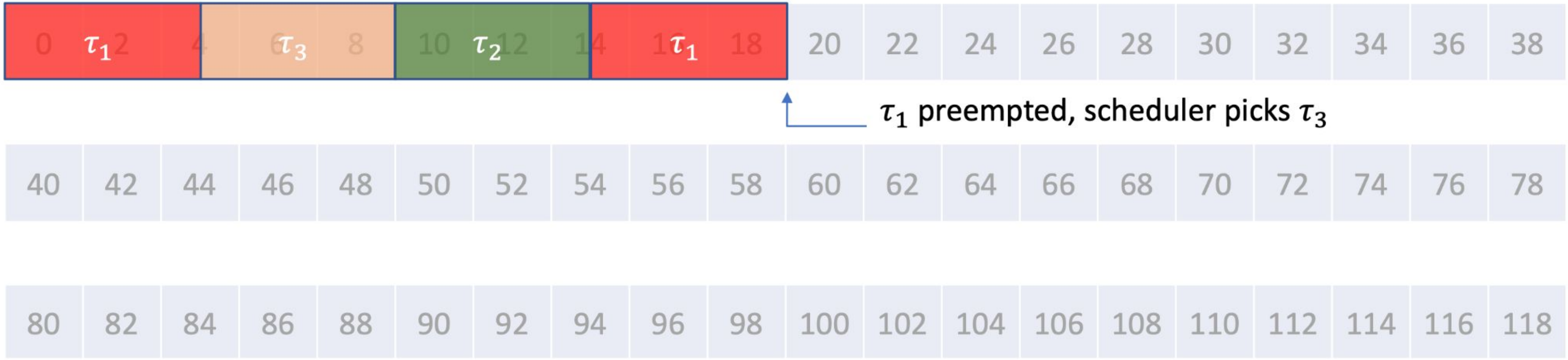


Round-robin Scheduling (Contd.)

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 (Contd.)

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: $\{\tau_1, \tau_3, \tau_2\}$

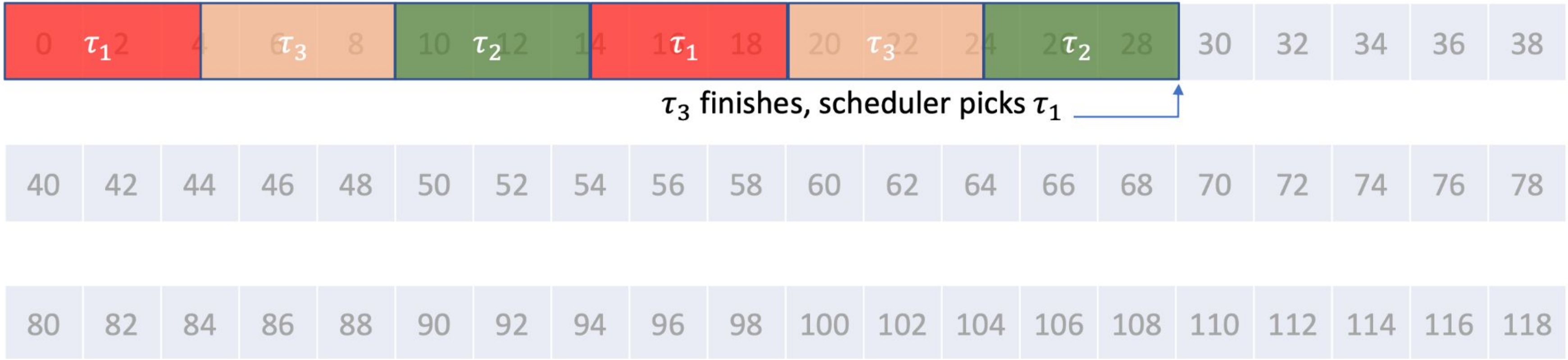
Round-robin Scheduling (Contd.)

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: $\{\tau_1, \tau_3\}$

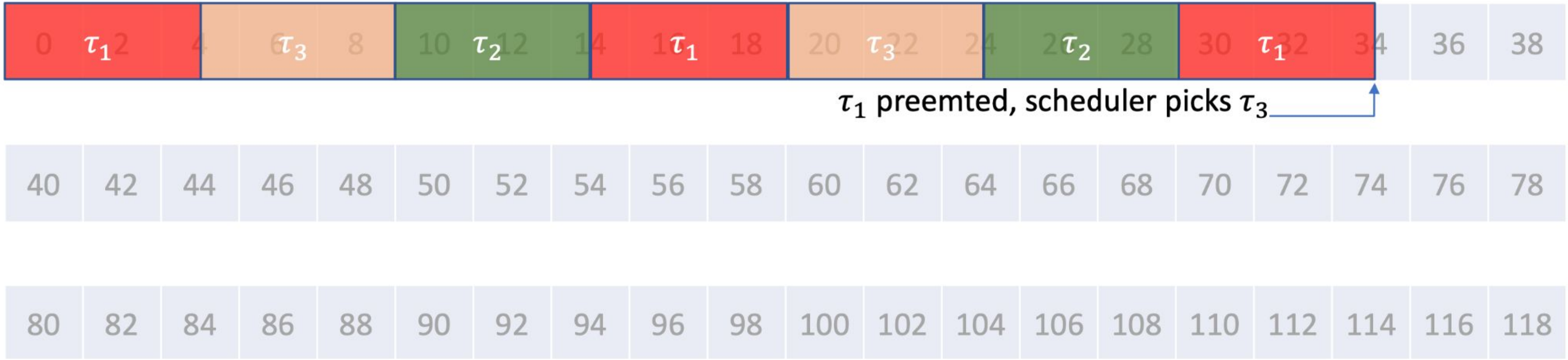


Round-robin Scheduling (Contd.)

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: $\{\tau_1, \tau_3\}$



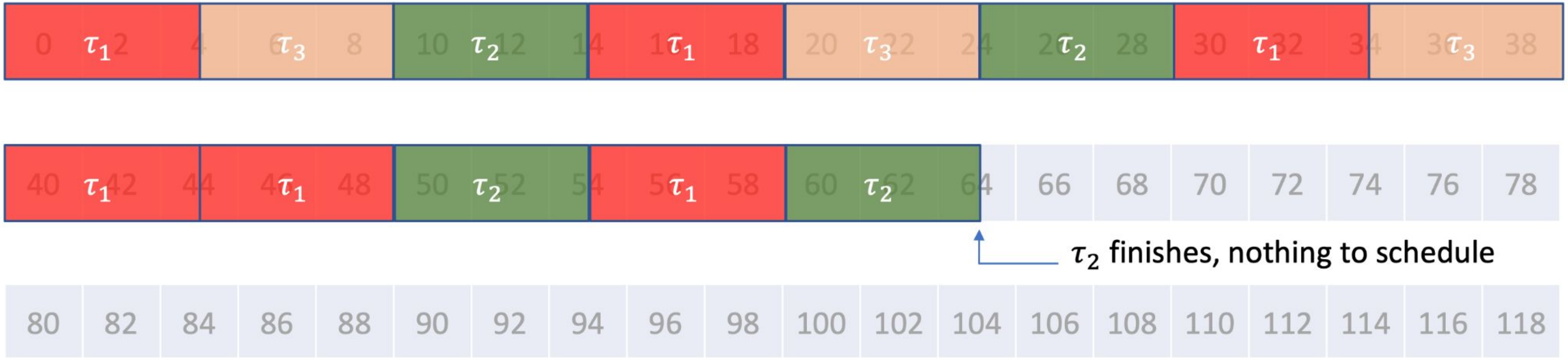
Round-robin Scheduling (Fast-forward)

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: Ø



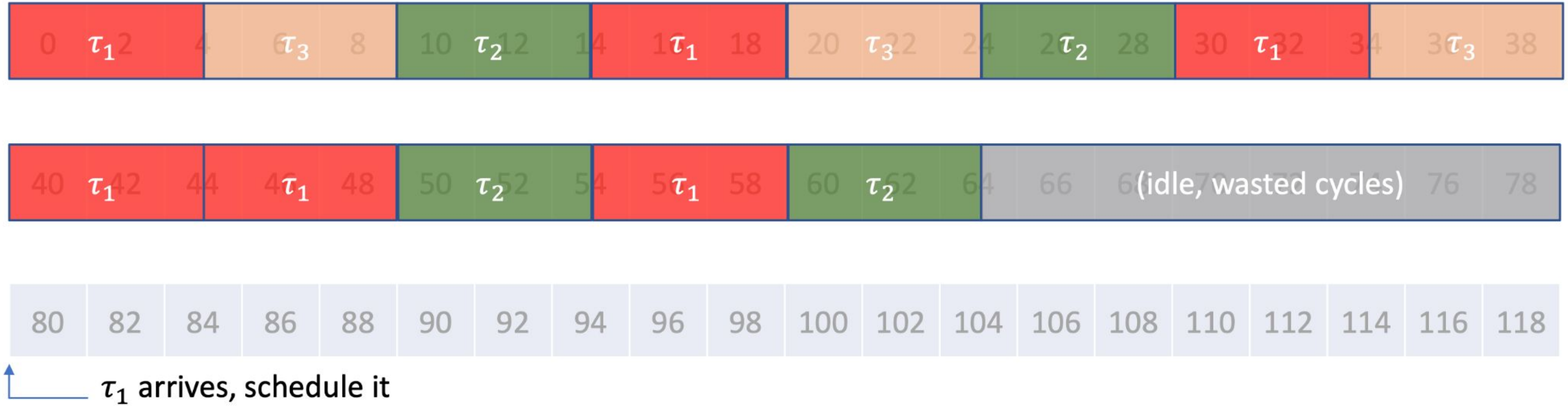
Round-robin Scheduling (Contd.)

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: $\{\tau_1\}$



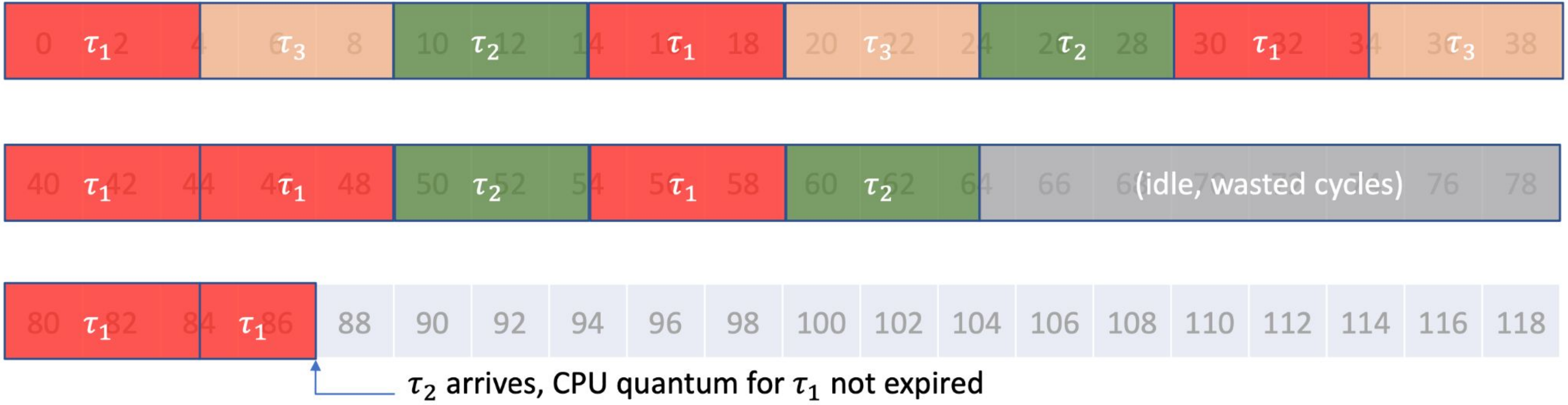
τ_1 arrives, schedule it

Round-robin Scheduling (Contd.)

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: $\{\tau_1, \tau_2\}$

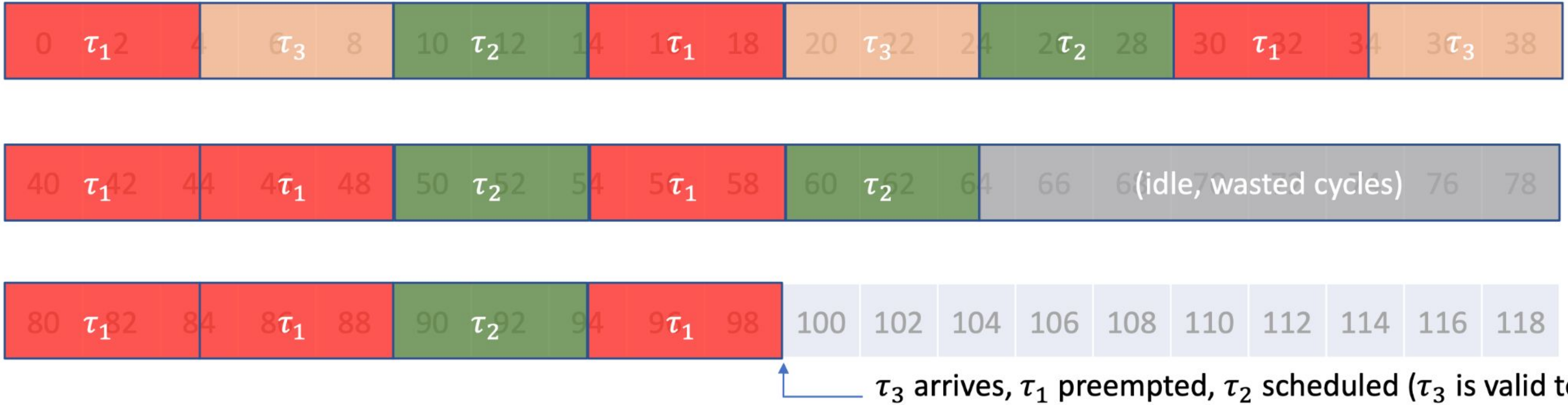


Round-robin Scheduling (Contd.)

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: $\{\tau_1, \tau_2, \tau_3\}$



τ_3 arrives, τ_1 preempted, τ_2 scheduled (τ_3 is valid tc

CPU Utilization

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

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

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

- CPU was is idle 22.5% of the time!
- Criteria for a good scheduler:
  - Max CPU utilization; Max throughput
  - Min turnaround time; Min waiting time; Min response time

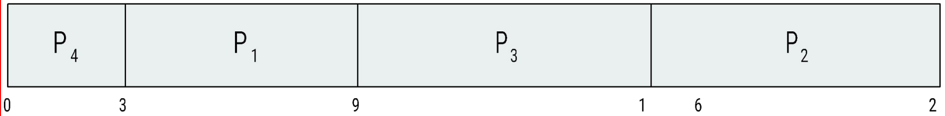
Other ‘Easy’ Scheduling Algorithms

First- Come, First-Served (FCFS)

- Waiting time for P1 = 0; P2 = 20; P3 = 10
- Average waiting time: (0 + 20 + 10)/3 = 10

Process	Arrival	Burst Time
P1	0	24
P2	4	3
P3	17	3

Shortest-Job-First (SJF)



- Average waiting time = (3 + 16 + 9 + 0) / 4 = 7

Process	Arrival	Burst Time
P1	0	6
P2	0	8
P3	0	7
P4	0	3

Rate Monotonic Scheduler

⇒ Real time priority scheduler

- The task with the shortest ***period*** is scheduled first
- Task is run until it finishes
- Running task can be preempted
  - But need to be with higher priority

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.

*Can you track the time-process horizon? (aka **Gantt chart**)*

Rate Monotonic Scheduler

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

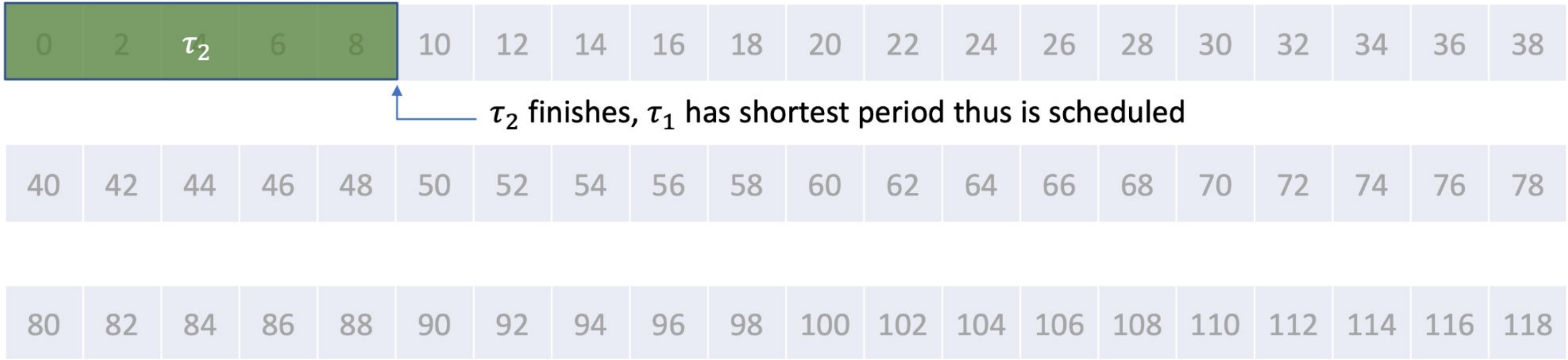
40	42	44	46	48	50	52	54	56	58	60	62	64	66	68	70	72	74	76	78

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

Rate Monotonic Scheduler

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 Scheduler (Fast Forward)

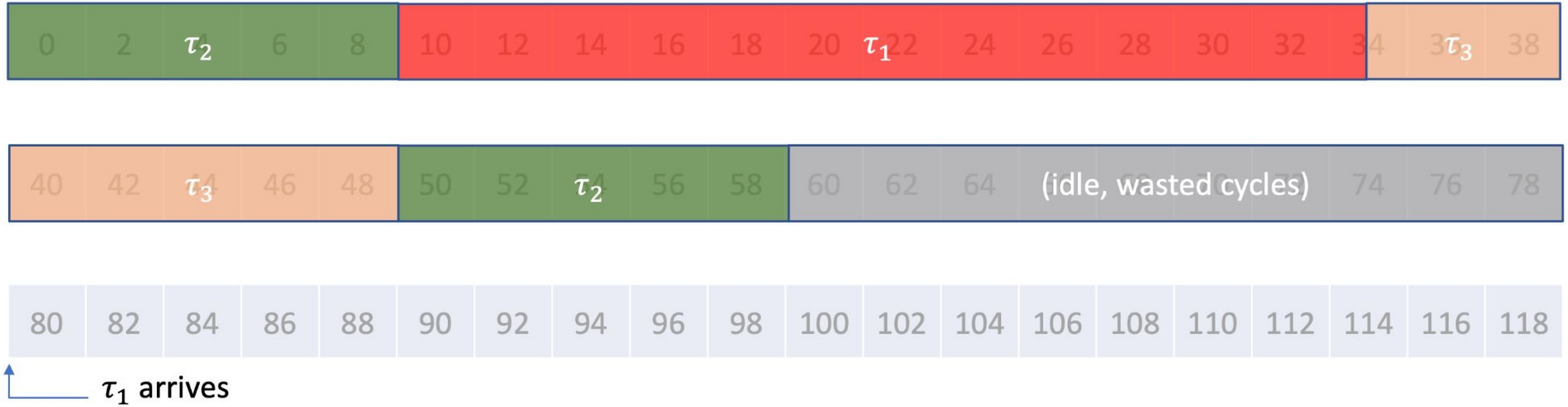
Task	Period	Arrival	Burst Length
τ_1	80	0	25
τ_2	50	0	10
τ_3	100	0	15

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

Rate Monotonic Scheduler (Contd.)

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

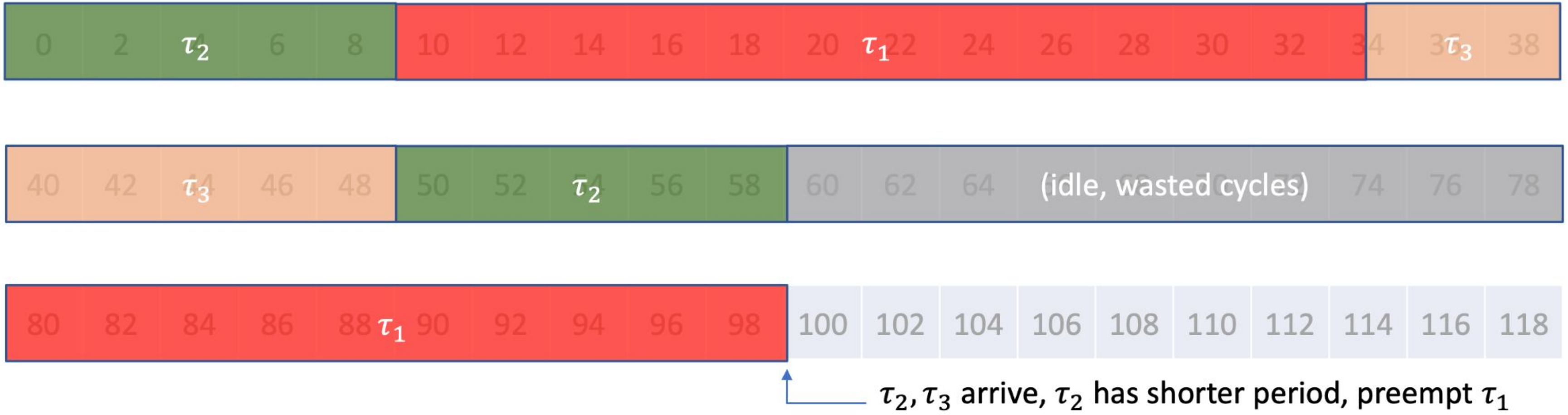
Run Queue at T = 80: $\{\tau_1\}$



Rate Monotonic Scheduler (Contd.)

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

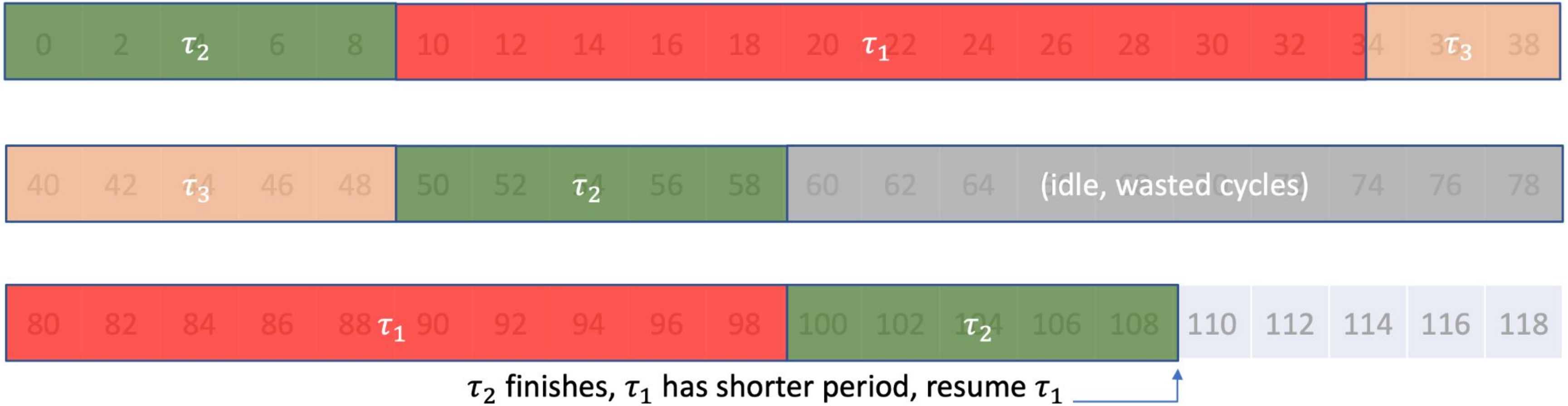
Run Queue at T = 100: $\{\tau_1, \tau_2, \tau_3\}$



Rate Monotonic Scheduler (Contd.)

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

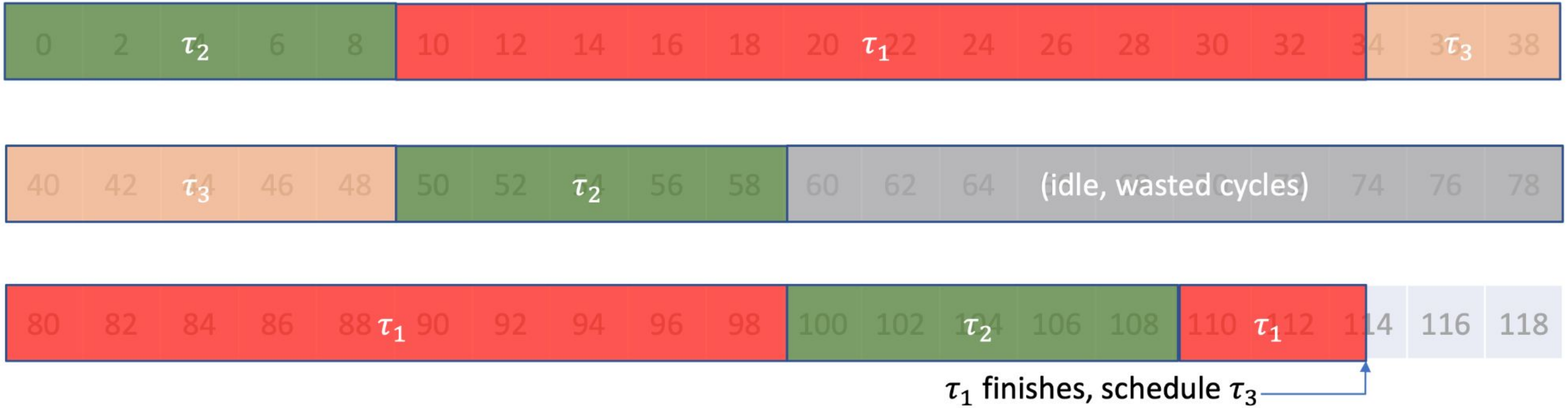
Run Queue at T = 110: $\{\tau_1, \tau_3\}$



Rate Monotonic Scheduler (Contd.)

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: $\{\tau_3\}$



Deadline Monotonic Scheduling

⇒ Real time priority scheduler

- Also known as **Earliest Deadline First (EDF)** scheduling
- The task with the earliest ***deadline*** is scheduled first
  - Important: pay attention to the deadline *in a given period*
- Running task can be preempted
  - But need to be with higher priority

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.

Can you track the time-process horizon?

*(aka **Gantt chart**)*

Deadline Monotonic Scheduling

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

Task-1 deadline in this cycle: 80

Task-2 deadline in this cycle: 50

Task-3 deadline in this cycle: 100

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 (Contd.)

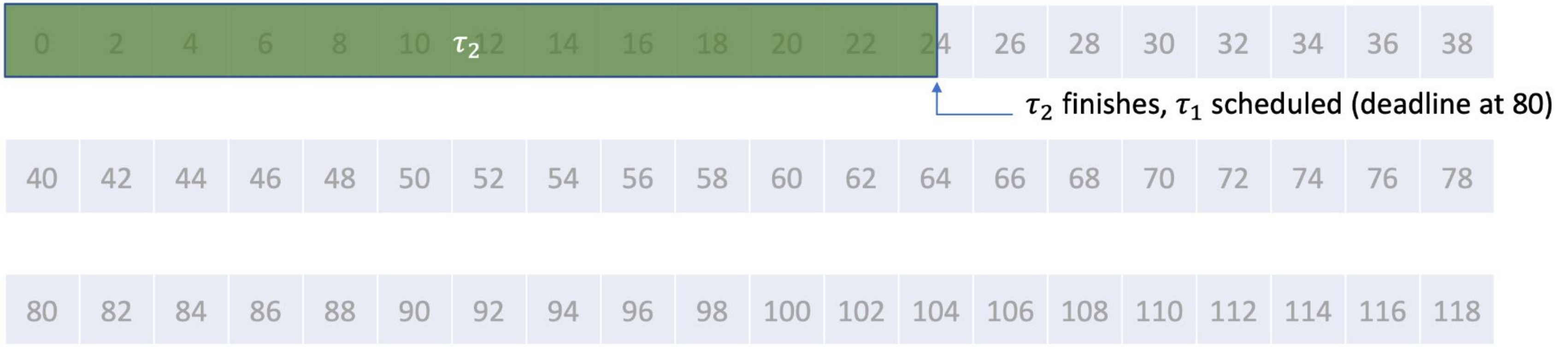
Task	Period	Arrival	Burst Length
τ_1	80	0	20
τ_2	50	0	25
τ_3	100	0	15

Task-1 deadline in this cycle: 80

Task-2 deadline in this cycle: Done

Task-3 deadline in this cycle: 100

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



Deadline Monotonic Scheduling (Contd.)

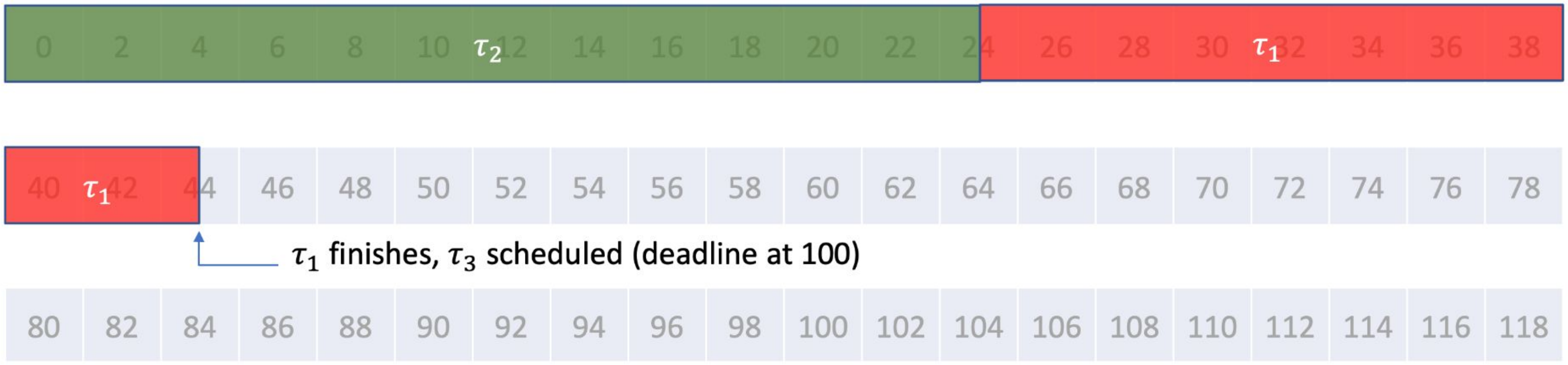
Task	Period	Arrival	Burst Length
τ_1	80	0	20
τ_2	50	0	25
τ_3	100	0	15

Task-1 deadline in this cycle: Done

Task-2 deadline in this cycle: Done

Task-3 deadline in this cycle: 100

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



Deadline Monotonic Scheduling (Contd.)

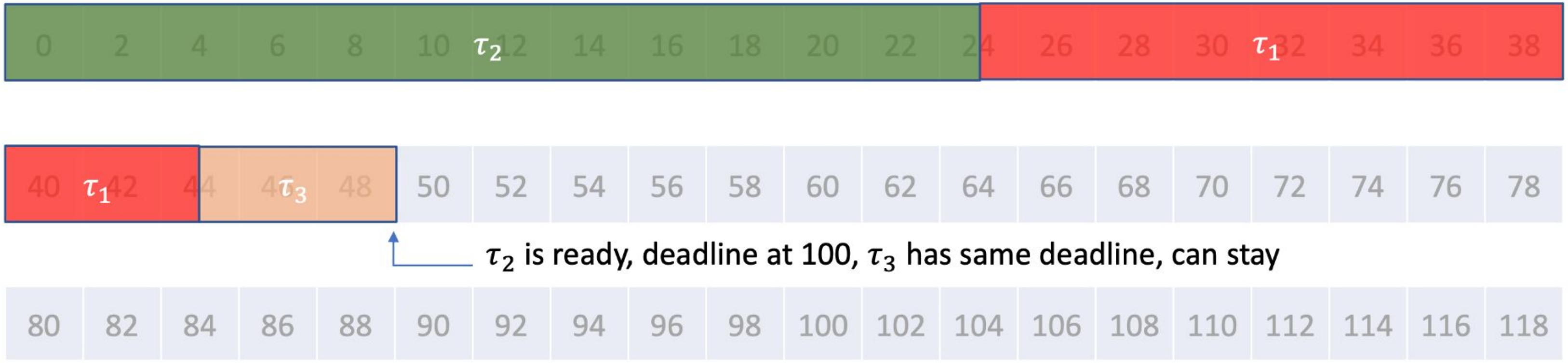
Task	Period	Arrival	Burst Length
τ_1	80	0	20
τ_2	50	0	25
τ_3	100	0	15

Task-1 deadline in this cycle: Done

Task-2 deadline in this cycle: 100

Task-3 deadline in this cycle: 100

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



Deadline Monotonic Scheduling (Contd.)

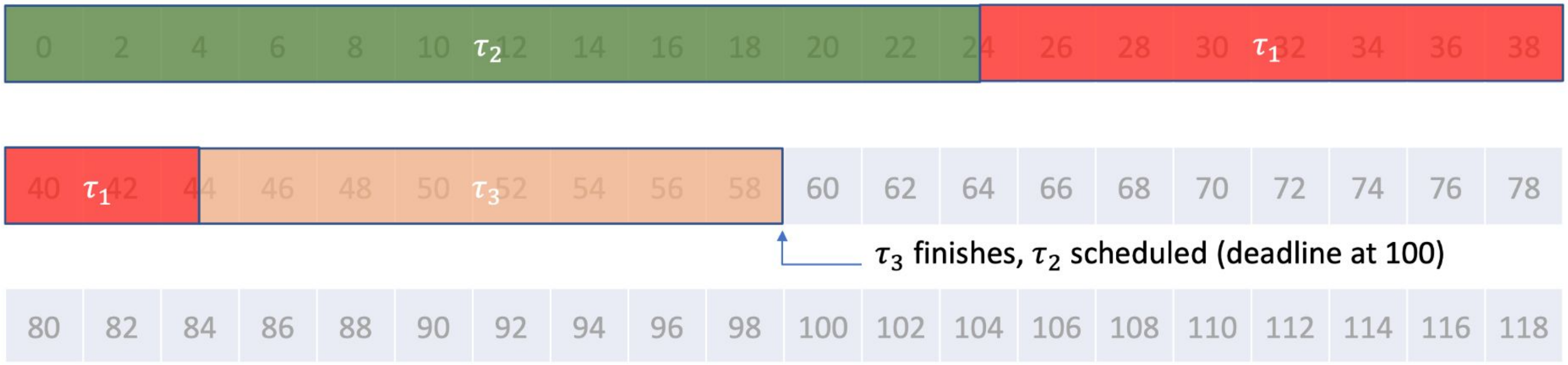
Task	Period	Arrival	Burst Length
τ_1	80	0	20
τ_2	50	0	25
τ_3	100	0	15

Task-1 deadline in this cycle: Done

Task-2 deadline in this cycle: 100

Task-3 deadline in this cycle: Done

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



Deadline Monotonic Scheduling (Contd.)

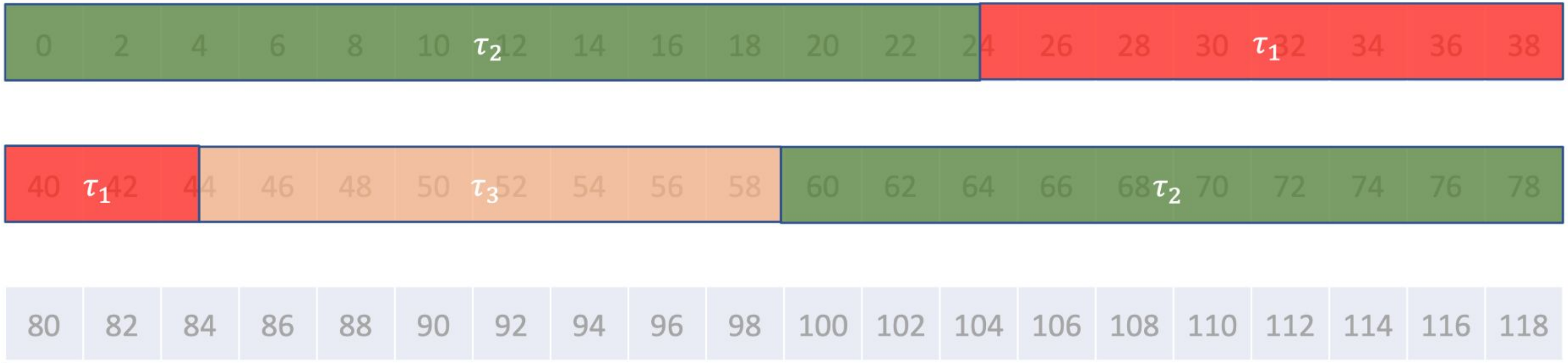
Task	Period	Arrival	Burst Length
τ_1	80	0	20
τ_2	50	0	25
τ_3	100	0	15

Task-1 deadline in this cycle: 160

Task-2 deadline in this cycle: 100

Task-3 deadline in this cycle: Done

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

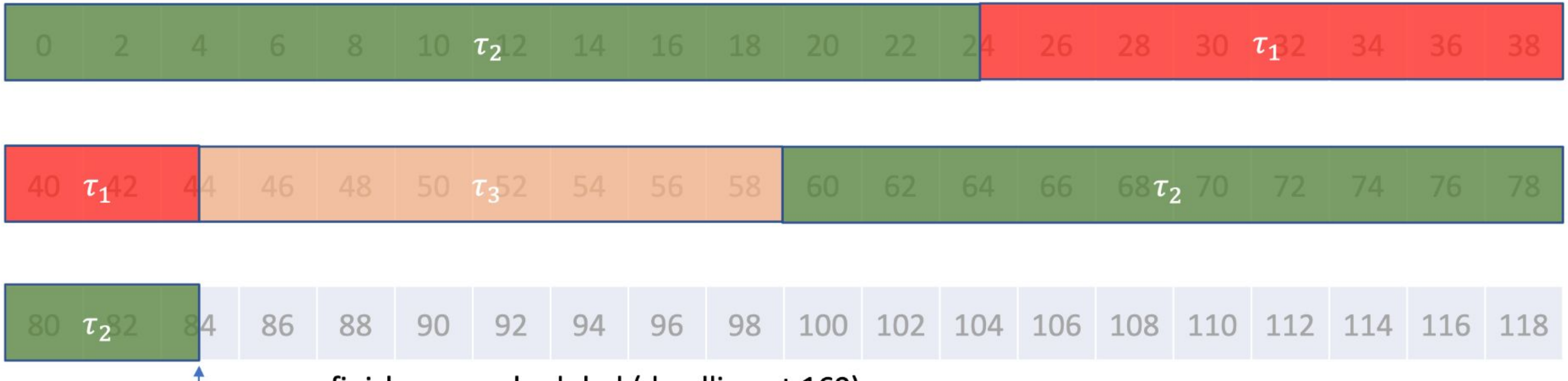


τ_1 arrives (deadline at 160), τ_2 remains (deadline at 100)

Deadline Monotonic Scheduling (Contd.)

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: $\{\tau_1\}$

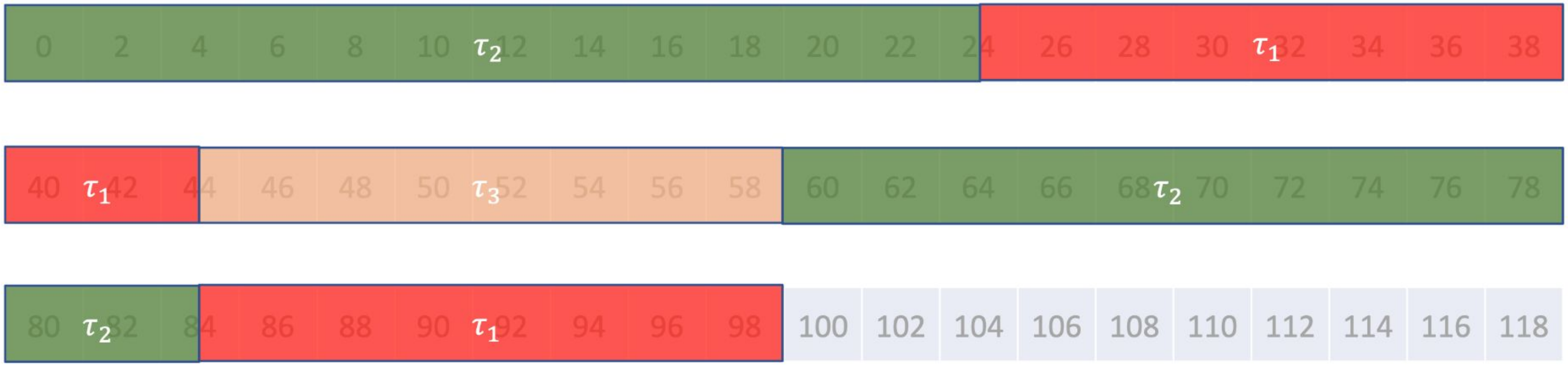


τ_2 finishes, τ_1 scheduled (deadline at 160)

Deadline Monotonic Scheduling (Contd.)

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: $\{\tau_1, \tau_2, \tau_3\}$



τ_2, τ_3 arrive (150, 200), τ_1 (160) preempted by τ_2

Other Important OS Components

- **File management and sharing**
  - Disk handler, meta-data handler
  - Database and file systems
- **Protection**
  - Network access and security; hardware security
- **Memory management**
  - Disk, memory access, and cache management
  - Object management: signaling and buffering
  - Advanced scheduling and deadlock management
- **I/O management and interfacing**
  - On-board hardware and peripherals; external devices
- **Specialized OS:** ROS - Robot Operating System

Sun	HP	IBM	VMWare	Apple	Oracle
OS X	Windows	Linux	Xen		
Red Hat	Fedora	CentOS	Debian	Ubuntu	Mint
SUSE	Mageia	Arch Linux	Slackware	Mandriva	Gentoo
FreeBSD	OpenBSD	NetBSD	DragonFly BSD	Darwin	

About 40-50 years of advanced literature. Now we know where to start!