Embedded Operating Systems

What is an Embedded OS?

- Operating systems that run on embedded devices/systems ☺
- Examples of Embedded systems
  - Smart watches/digital watches, air-conditioners, home security systems, microwave ovens, washing machines, traffic light controllers, automotive systems (braking system, entertainment system, fuel injection systems,…), routers, IoT devices, POS terminals, vending machines, photocopiers, medical devices (pacemakers, heart rate monitors, …), phones, missile guidance systems, …..

- General characteristics of embedded systems
  - Special purpose systems
    - Designed to do a specific set of tasks only
  - Interacts with the environment in the form of reacting to events, user input etc.
    - However, most do not need extensive UI support
  - Needs some real time guarantees for
    - Safety (Ex.: Fire alarm system, Automobile braking system, Pacemaker,…))
    - Usability (Ex.: Washing machine, Smartwatch, Photocopier….)
  - Can be resource constrained
    - Less CPU power, memory, storage etc.
  - Power may be an issue

- How to write software for embedded systems
  - Approach 1: Directly write the application to run on the target device
    - Write the application in a high level language using maybe an IDE
      - Can have library support also
    - Cross-compile to generate image to run on target hardware
    - Load the image on the target hardware
    - Advantages:
      - No extra code, so can make it very efficient
      - Very good for resource constrained devices running small applications
    - Disadvantages
      - Not suitable for complex applications on more complex hardware
      - Costly to develop and maintain

- Approach 2: Build services on top of an existing OS
  - Start with an existing OS for core functionalities like scheduling, IPC, memory management etc.
  - Use build/configuration tools to choose services/packages/modules that are necessary for the application at hand on top of the core OS
  - Build the image with only what is chosen
  - Advantages:
    - Faster development as does not need to start from scratch and use existing modules
    - Open source collaborations can give a very wide range of services/modules to choose from
    - More reliable as tested OS used
  - Disadvantages
    - Core OS may not have all things needed for an application (Ex.: limited scheduling policy options, power management, …)
    - May need more resources on the target device
    - Less efficient due to added layer
  - Example: Ubuntu Core, Yocto, Buildroot, Windriver Linux, …

- Approach 3: Build an OS customized for embedded systems
  - By changing an existing OS or writing from scratch
  - Advantages:
    - Obvious
  - Disadvantages:
    - Usually commercial, more costly
  - Example: VxWorks, QNX, INTEGRITY, Litmus-RT, …

Real Time Scheduling

- Real Time Tasks
  - Tasks that need time-bounded response/completion
  - Every task has an arrival (ready) time, an execution time, and a deadline
  - Hard real time tasks
    - Task must complete before or at deadline.
    - Value of completing the task after deadline is zero
  - Soft real time tasks
    - Missing deadline incurs a penalty
    - Penalty increases as duration by which deadline is missed increases

- Examples
  - Hard real-time tasks
    - Automotive braking system
    - Missile guidance system
    - Accident warning system
    - ….
  - Soft real-time tasks
    - Washing machines
    - Photocopiers
    - Automotive in-vehicle entertainment system
    - ….

- Tardiness
  - Completion Time – Deadline (lower bounded to 0)
  - Goal is to have 0 tardiness, or at least predictable tardiness
- Laxity
  - Deadline – Remaining execution time
  - This gives the maximum duration the task can be delayed so that it still meets its deadline
- Response Time
  - Completion time – Arrival time
- Periodic tasks
  - Tasks that repeat after a known period
- Aperiodic tasks
  - Can arrive at any time

- *Question: Given a set of tasks with known arrival times, execution times, and deadlines, how do we schedule them?*
- Feasible schedule
  - A schedule in which all tasks complete while meeting a set of specified constraints
    - Ex: All tasks meet their deadlines
- A set of tasks is schedulable if there exists a feasible schedule for the set
- Optimal scheduling algorithm
  - One that always finds a feasible schedule for a given set of tasks, if one exists

- What performance measures are we interested in for a group of tasks as a whole?
  - Average response time
  - Makespan (total completion time)
  - Average/Maximum tardiness
  - Number of tasks that miss their deadlines
  - …

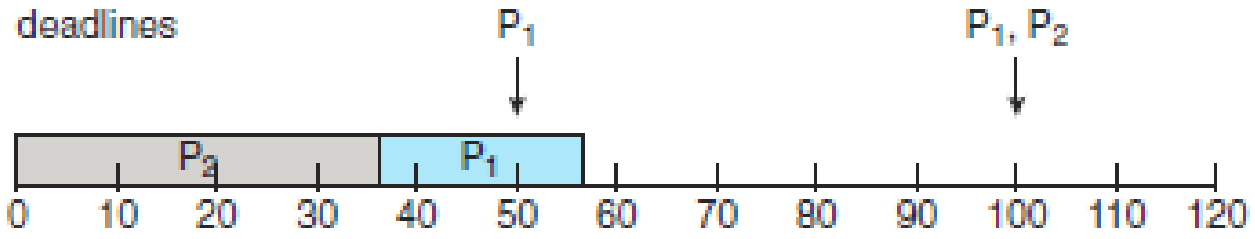
- Challenges in finding feasible schedules
  - Meeting deadlines and achieving fairness can be contradictory
  - Uncertainty in the system (interrupts, migrations, other tasks) can affect the guarantees
- Things that work in your favor
  - Hard real time guarantees are not needed always
  - Bounds on task execution times are known in many cases
  - Task arrival times are known in many cases
  - Fairness is not important in many cases
    - Low priority tasks may have little consequences
  - Uncertainties can be mitigated
    - Disable interrupts for critical tasks, use processor affinity to stop migrations, dedicated system, …

Types of Scheduling Algorithms

- Static scheduling
  - Schedule is fixed a-priori statically, no scheduling decision during runtime
- Fixed priority scheduling
  - Each process assigned a fixed priority at runtime
  - Scheduler schedules as per priority
  - Examples: RT-priority scheduling in Linux, Rate Monotonic Scheduling, …
- Dynamic priority scheduling
  - Process priority is a function of the current state of the system and can change
  - Examples: Earliest Deadline First, Least Slack Time First, …

Fixed Priority Scheduling

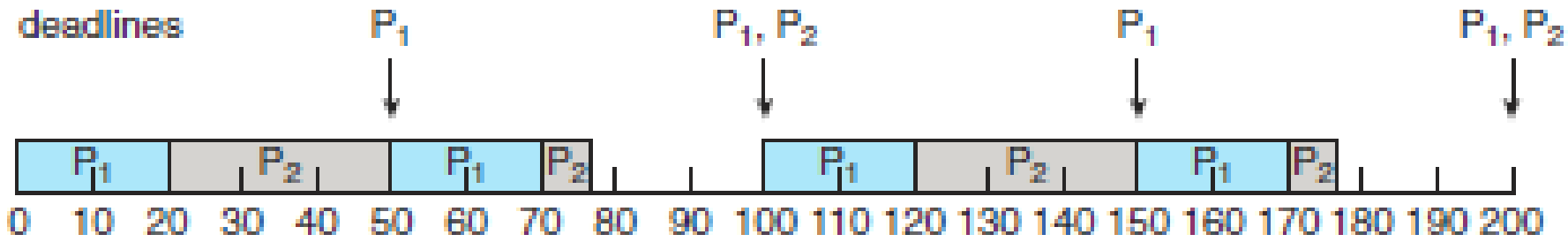
- Why do we need to assign priorities in a proper manner?
  - Consider two periodic tasks T1 and T2 with
    - T1: execution time = 20, period = 50, relative deadline = 50
    - T2: execution time = 35, period = 100, relative deadline = 100
  - Arbitrary priority assignment
    - T1 > T2, or
    - T2 > T1 (T1 will miss deadline)



Rate Monotonic Scheduling

- Fixed Priority scheduling
- Set of periodic tasks, each with a period and execution time
- Deadline is equal to the period
  - Current invocation must finish before the next time the task is invoked
- Priority assigned based on period
  - Higher priority to tasks with shorter periods and vice-versa

- T1: execution time = 20, period = 50, relative deadline = 50
- T2: execution time = 35, period = 100, relative deadline = 100



Deadline Monotonic Scheduling

- Fixed Priority scheduling
- Set of periodic jobs, each with a period and execution time
- Deadline is less than or equal to period
- Priority assigned based on deadline
  - Higher priority to processes with lower deadlines

- Rate Monotonic and Deadline Monotonic scheduling algorithms are both non-optimal
  - T1: execution time = 25, period = 50, relative deadline = 50
  - T2: execution time = 35, period = 80, relative deadline = 80
- Deadline will be missed for the above task set irrespective of how you prioritize them
  - But it has a feasible schedule
- Known result: No fixed priority scheduling algorithm can be optimal

- Known result: Rate Monotonic Scheduling algorithm will always find a feasible schedule for a set of periodic tasks with relative deadlines equal to their respective periods if a feasible schedule can be found by any other static priority algorithm.
- Same is true for Deadline Monotonic Scheduling algorithm when relative deadline is less than or equal to the period

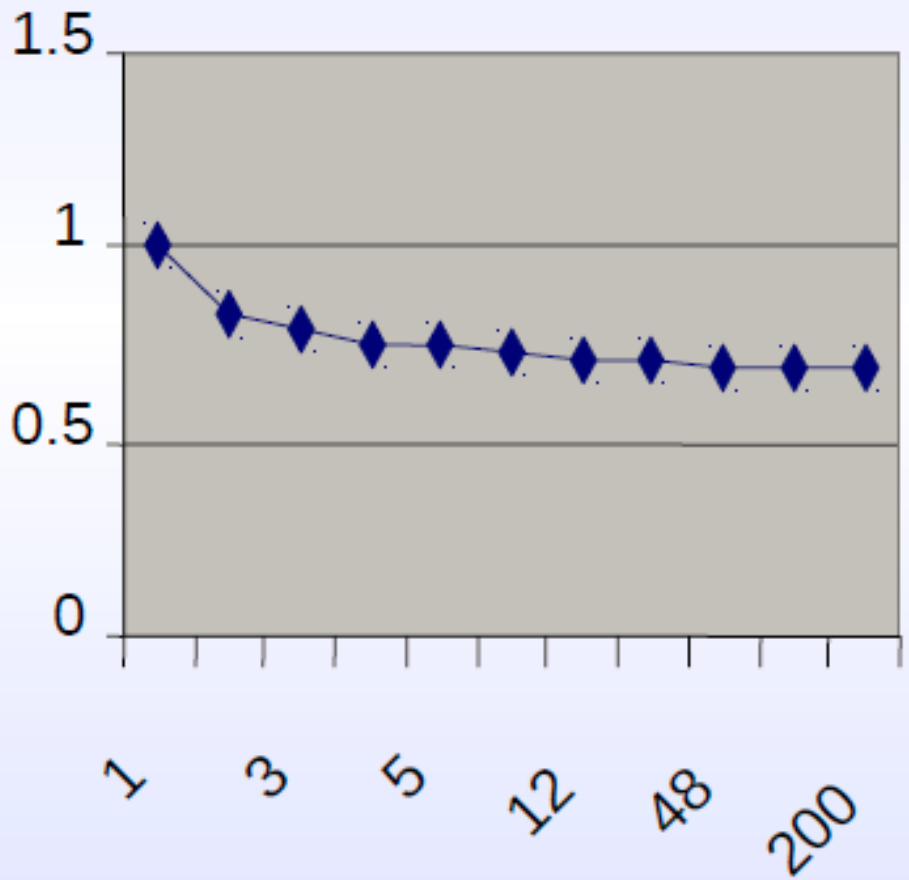
- Schedulability Test
  - Can we test if a given set of tasks have a feasible schedule?
  - Does not mean we will find it, just exists or not
- Sufficiency Tests
  - If test is passed, there exists a feasible schedule
  - If not passed?
- Necessity Test
  - If test is not passed, there does not exist any feasible schedule
  - If passed?

- Schedulability test for Rate Monotonic scheduling
  - Consider a set of tasks $T_1, T_2, T_3, \ldots T_n$, with execution times $C_1, C_2, \ldots C_n$, and periods $P_1, P_2, P_3, \ldots P_n$
  - Then, there exists a feasible schedule if

$$\sum \frac{C_i}{P_i} \leq n(2^{1/n} - 1)$$

  - Preemptive Rate Monotonic Scheduling is optimal in this case, it will always find a feasible schedule
- So, given a set of periodic hard real time tasks
  - Do the test
  - If it passes, apply Rate Monotonic Scheduling, deadlines will always be met

m	$m*(2^{1/m}-1)$
1	1
2	0.828427125
3	0.77976315
4	0.75682846
5	0.743491775
6	0.73477229
12	0.713557132
24	0.703253679
48	0.698176077
96	0.695655573
200	0.694349702

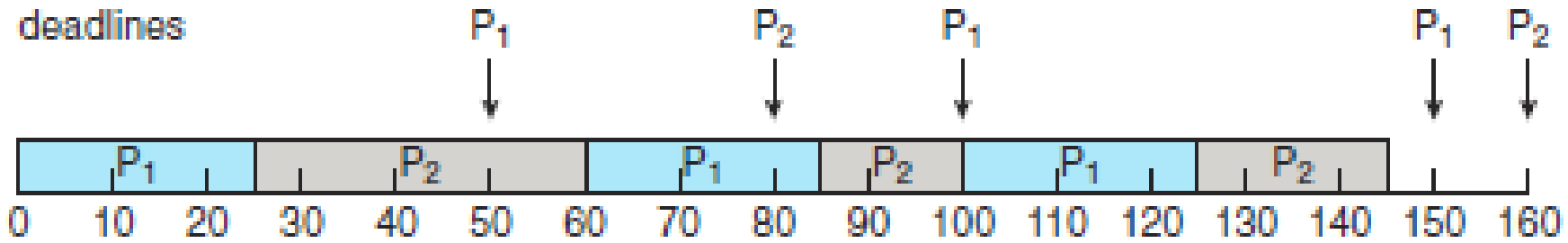


- Intuition
  - (Execution time/Period) gives fraction of time the task keeps the CPU utilized
  - If the sum of the times the CPU is occupied is less, more chance of meeting deadlines for all

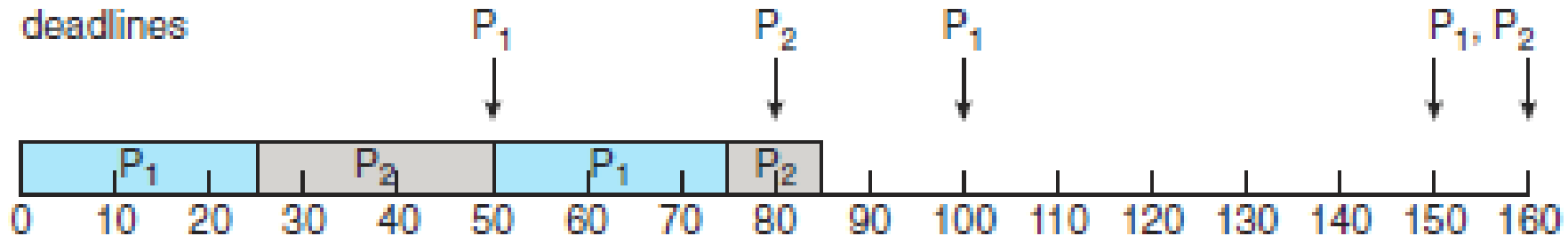
Earliest Deadline First Scheduling

- Assign each task a priority based on how close the absolute deadline is
- Scheduler schedules the task that is closest to its deadline
- Preemptive dynamic priority scheduling method

  - T1: execution time = 25, period = 50, relative deadline = 50
  - T2: execution time = 35, period = 80, relative deadline = 80



- This would have missed deadline with Rate Monotonic Scheduling



- Utilization sum = 0.9375

- EDF does not require tasks to be periodic
- EDF is optimal
- Schedulability tests for EDF
  - A set of periodic tasks with relative deadlines equal to their periods can be feasibly scheduled with EDF *if and only if* its total utilization is at most one
  - There is a similar result when deadline is less than or equal to period but bound is different

Embedded Linux

- Embedded Linux based system is one that uses Linux as the operating system in the embedded system ☺
- Built from same Linux kernel from kernel.org
  - There is no separate embedded version shipped of the Linux kernel
- But packages/services/modules can be customized to fit the needs of the specific embedded hardware and application at hand.
- So essentially
  - Some common parts same as desktop Linux
  - Other parts removed/added as per the target hardware and application at hand.

- A comparison
  - Ubuntu 22.04.3 LTS Desktop
    - Minimum system requirement: 4 GB RAM, 25 GB disk, 2 GHZ dual core CPU
    - Around 4.7 GB image size
    - A very large part of the codebase is drivers, multiple user interfaces etc., which are not needed or not possible to run on smaller devices
  - Ubuntu Core
    - Ubuntu's version for IoT and embedded devices
    - Min: 384 MB RAM, 512 MB storage
    - Pre-built images are also available for download for a range of target hardware like Raspberry Pi, Intel NUC etc.

Ubuntu Core

- Different modules called snaps
  - Kernel, root file system, bootloader, audio, disk, Bluetooth, networking, …
- Snaps can be published by developers for different applications and browsed and downloaded
  - Open source collaboration enables large number of available snaps for many things
- So can configure the base system (which is also built from a set of snaps) with what is needed

Other Open Source Build Environments

- Yocto, Buildroot
  - Unlike Ubuntu Core, not an embedded Linux Distribution, but allows you to create one
  - Allows custom Linux distributions to be built for different hardware and different needs
  - Provides toolsets and development environment for creating custom embedded Linux images for specific target hardware and applications
  - Yocto has lot more flexibility than Buildroot, but is more complex to use also

Commercial Embedded OS

- VXWorks
  - Gives OS, build environment customization (VxWorks Studio) for optimized image generation for target hardware, and application development (Workbench) support
  - Windriver, the company behind VxWorks, also gives WindRiver Linux
    - A development environment for creating customized Linux images for embedded systems
    - Built on Yocto project
      - More like Yocto project plus support plus additional development tools etc.
- QNX
- INTEGRITY
- LynxOS

Litmus-RT

- A research OS that adds real time options to Linux kernel
  - Focus on scheduling real time tasks on multiprocessors
  - Adds support for multiple real time scheduling algorithms such as
    - Partitioned EDF (PSN-EDF)
    - Global EDF(GSN-EDF),
    - Clustered EDF (C-EDF),
    - Partitioned Fixed-Priority (P-FP),
    - …
  - Allows support for plugging in your own scheduling policies
  - Not really focused specifically on embedded systems, but still useful
  - No longer supported (last stable version in 2017), but still used a lot in academics

Are we forgetting something??