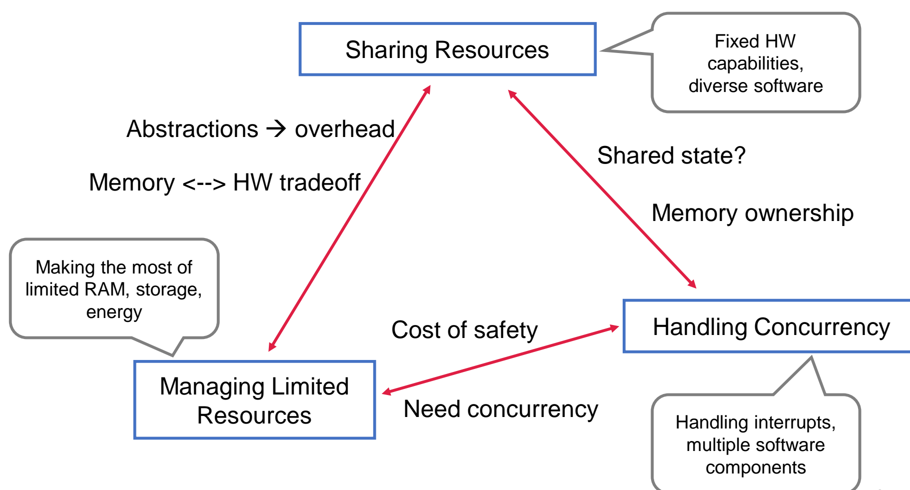
Embedded Operating Systems

Embedded OS: Three Key Requirements



Embedded hardware characteristics

- Resource constrained
  - 10s KB RAM
  - 100s KB flash
  - 50-100 MHz CPU
  - Limited energy
- Hardware
  - Single core
  - No MMU
  - Heavily relies on interrupts
- Deployment
  - Unattended operation
  - Many separate operations

This week

- How do OSes differ on resource constrained devices deployed in the physical world?
- Challenges
  - Limited HW capability, time, energy
  - Limited space to store OS code
  - No virtual memory
  - No user monitor
- Opportunities
  - Simpler hardware
  - Fixed tasks?
  - Relaxed latency requirements?

Embedded General Purpose OS Exploration

- Two embedded operating systems
  - TinyOS
  - Tock
- Three questions
  - How do they enable resource sharing?
  - How do they manage concurrency?
  - How do they manage limited resources?

5

TinyOS

6

TinyOS solution

- Support concurrency
  - event-driven architecture
- Software modularity
  - application = scheduler + graph of components
  - A component contains commands, event handlers, internal storage, tasks
  - nesC language to facilitate this
- Efficiency: get done quickly and then sleep
- Static memory allocation

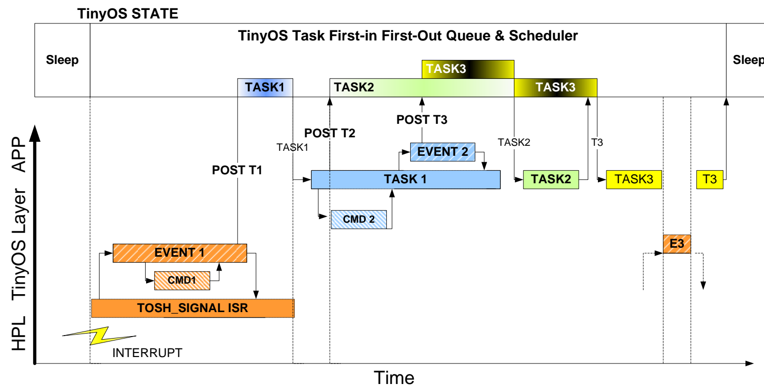
7

TinyOS computational concepts

1. Events
   - Time critical
   - Caused by interrupts (Timer, ADC, Sensors)
   - Short duration
2. Commands
   - Request to a component to perform service (e.g, start sensor reading)
   - Non-blocking, need to return status
   - Postpone time-consuming work by posting a task (split phase w/ callback event)
   - Can call lower-level commands
3. Tasks
   - Time flexible (delayed processing)
   - Run sequentially by TOS Scheduler
   - Run to completion with respect to other tasks
   - Can be preempted by events

8

TinyOS Execution Model



Concurrency

- Two threads of execution
  - Tasks
    - deferred execution
    - tasks cannot preempt other tasks
  - Hardware event handler: respond to interrupts
    - Interrupts can preempt tasks
- Scheduler
  - Two level scheduling
    - interrupts (vector) and tasks (queue)
  - Task queue is FIFO
  - Scheduler puts processor to sleep when no event/command is running and task queue is empty

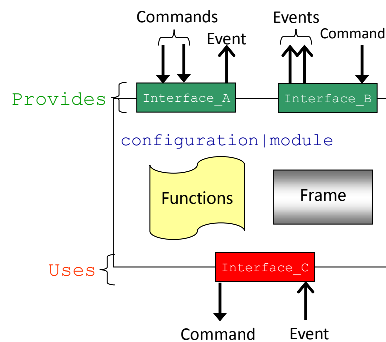
TinyOS Theory of Execution: Events & Tasks

- Consequences of an event
  - Runs to completion
  - Preempt tasks
- What can an event do?
  - **`signal`** events
  - **`call`** commands
  - **`post`** tasks
- Consequences of a task
  - No preemption mechanism
  - Keep code as small execution pieces to not block other tasks too long
  - To run a long operations, create a separate task for each operation, rather than using on big task
- What can initiate (post) tasks?
  - Command, event, or another task

12

Interface : provides & uses

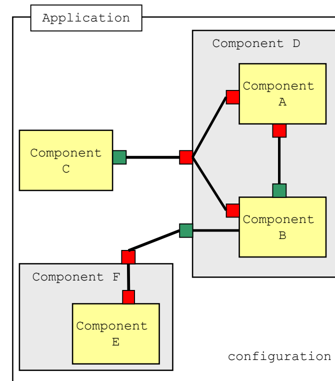
- Provides: Exposes functionality to others
- Uses: Requires another component
- Interface ***provider*** must implement commands
- Interface ***user*** must implement events



13

Applications consist of components “wired” together

- Application: graph of components
  - Main component
    - init, start, stop
    - first component executed
  - Other components
- Components
  - modules
  - configurations
- Interfaces: point of access to components
  - **uses**
  - **provides**



Example: Blink Configuration

```
configuration Blink {
}
implementation {
  components Main, BlinkM, SingleTimer, LedsC;

  Main.StdControl -> SingleTimer.StdControl;
  Main.StdControl -> BlinkM.StdControl;

  BlinkM.Timer -> SingleTimer.Timer;
  BlinkM.Leds -> LedsC;
}
```

Example: Blink Module

```
module BlinkM {
  provides {
    interface StdControl;
  }
  uses {
    interface Timer;
    interface Leds;
  }
}
implementation {
  command result_t StdControl.init() {
    call Leds.init();
    return SUCCESS;
  }
   command result_t StdControl.start() {
    // Start a repeating timer that fires every 1000ms
    return call Timer.start(TIMER_REPEAT, 1000);
  }

  command result_t StdControl.stop() {
    return call Timer.stop();
  }

  event result_t Timer.fired() {
    call Leds.yellowToggle();
    return SUCCESS;
  }
}
```


TinyOS Compilation

- Number of resource uses known at compile time
  - Memory statically allocated
- All TinyOS files merged into one .c file
  - Existing C compilers then just have to compile one file
    - More optimization possibilities

TinyOS summary

- Component-based architecture
  - Provides reusable components
  - Application: graph of components connected by “wiring”
- Three computational concepts
  - Event, command, task
- Tasks and event-based concurrency
  - Tasks: deferred computation, run to completion and do not preempt each other
  - Tasks should be short, and used when timing is not strict
  - Events: run to completion, may preempt tasks
  - Events signify completion of a (split-phase) operation or events from the environment (e.g., hardware, receiving messages)
- Today
  - Not too much active development
  - NesC too much overhead
  - Hardware has improved

18

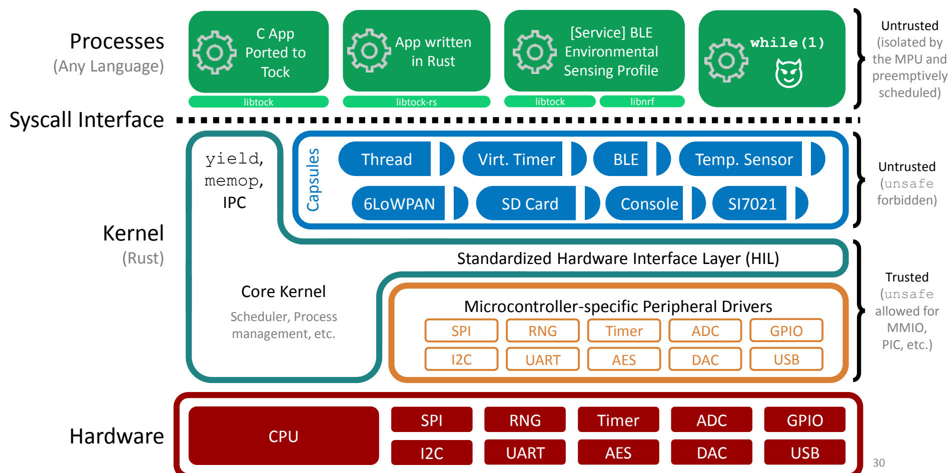
Tock

28

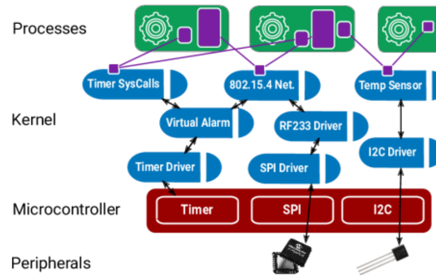
Tock: Embedded Devices are Multiprogrammed

- Multiple users running applications concurrently
- Applications updated dynamically
  - Small payloads better
  - Buggy updates shouldn't brick devices
- Security sensitive devices want *least privilege*

Syscall interface allows kernel to manage application access to resources



Different isolation primitives for an embedded OS



- **Processes**: Use the Memory Protection Unit
- **Capsules**: Type-safe Rust API for *safe* driver development
- **Grants**: Bind dynamic kernel resources to process lifetime

Tock's Isolation Mechanisms

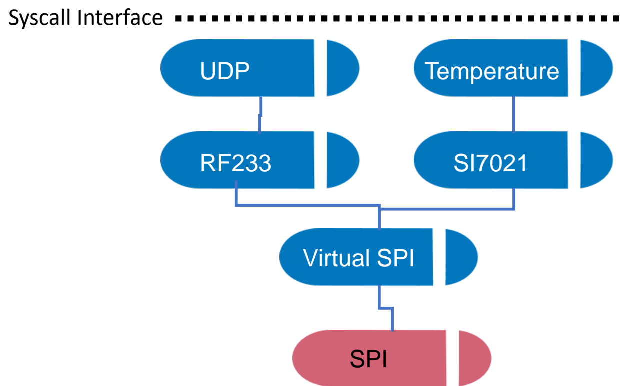
Totally untrusted

Processes
- Standalone executable in any language
- Isolation enforced at runtime
- Higher overhead
- Applications
- Time sliced

Trusted for liveness, not safety

Capsules
- Rust code linked into kernel
- Isolation enforced at compile-time
- Lower overhead
- Used for device drivers, protocols, timers...
- Cooperatively scheduled

Multiple capsules enable resource sharing



33

Multiple applications presents a new challenge: resource exhaustion

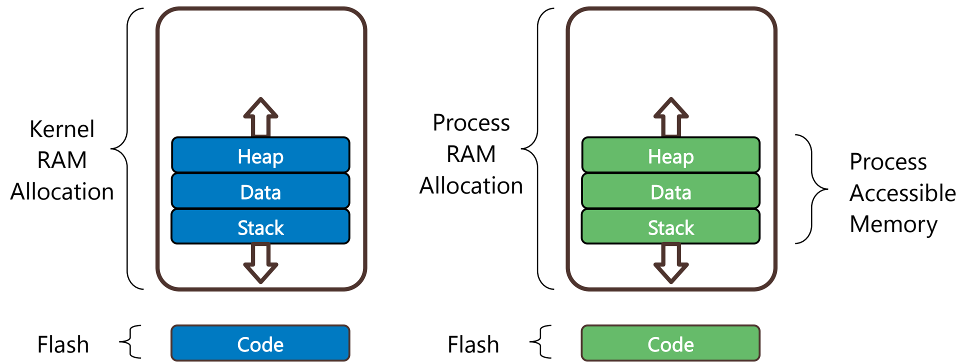
- How many userland processes will there be?

or

- How much memory should we allocate for them?
  - Or, should we do dynamic memory allocation?

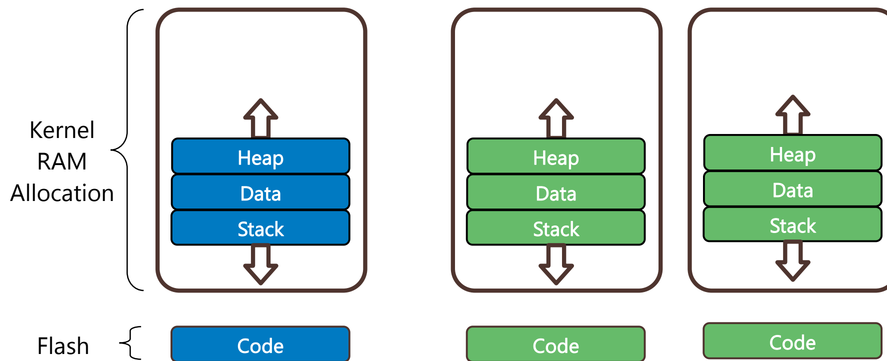
34

The kernel and processes have separate memory allocations. A kernel that allows allocations would have a heap, as shown.



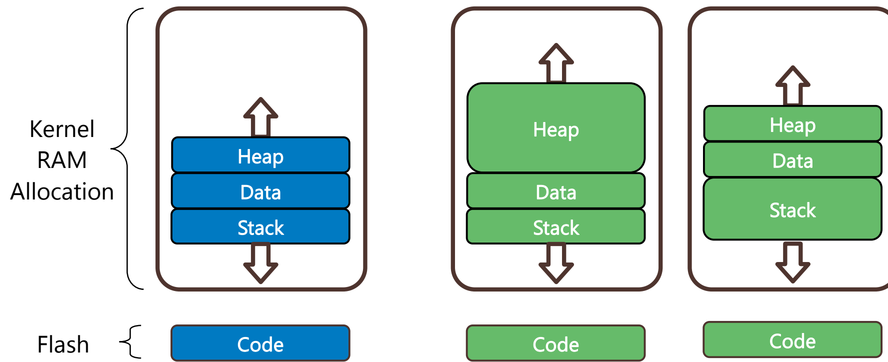
35

Tock supports multiple applications, each with their own memory allocations.



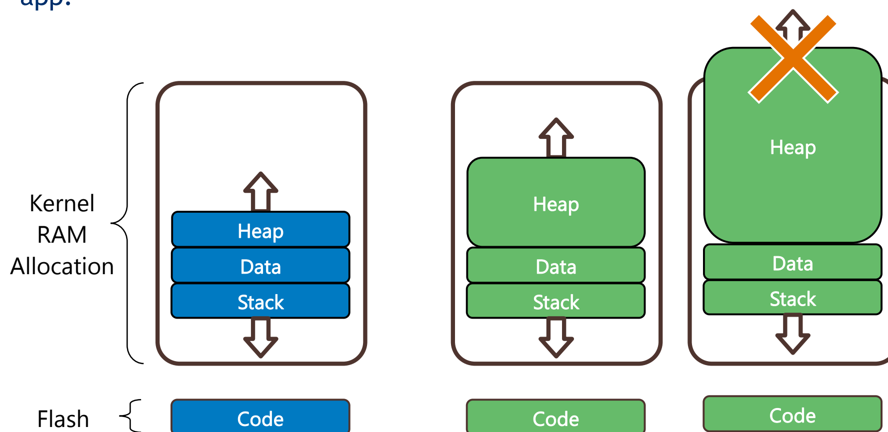
36

Apps are allowed to use their memory region as they see fit. For example, one app may need more heap space, another may need more stack space.



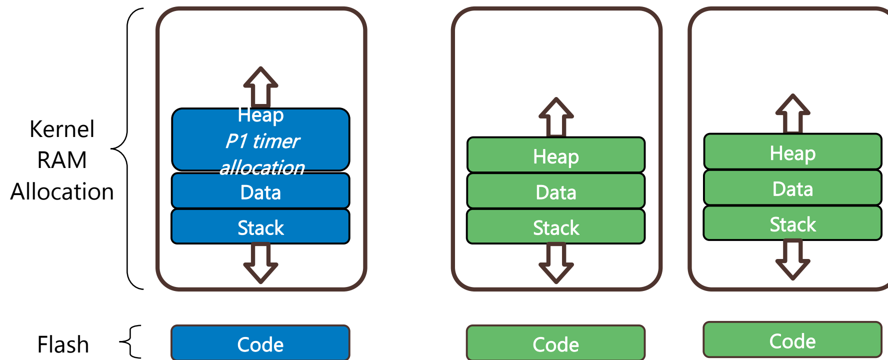
37

If one app exhausts its memory space, for example by making its heap too large, it will fault and the kernel can terminate just that app.

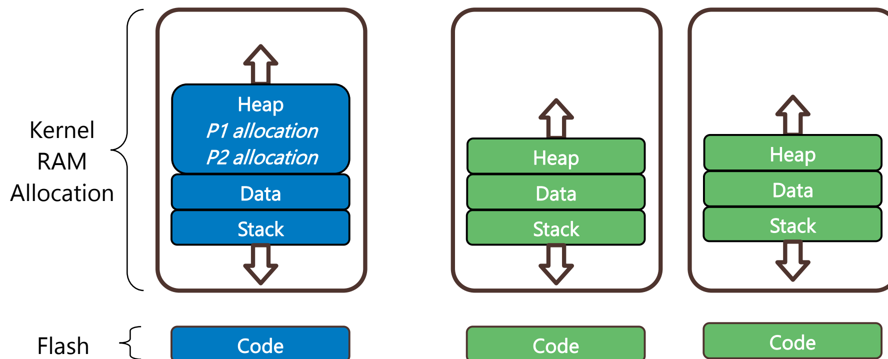


38

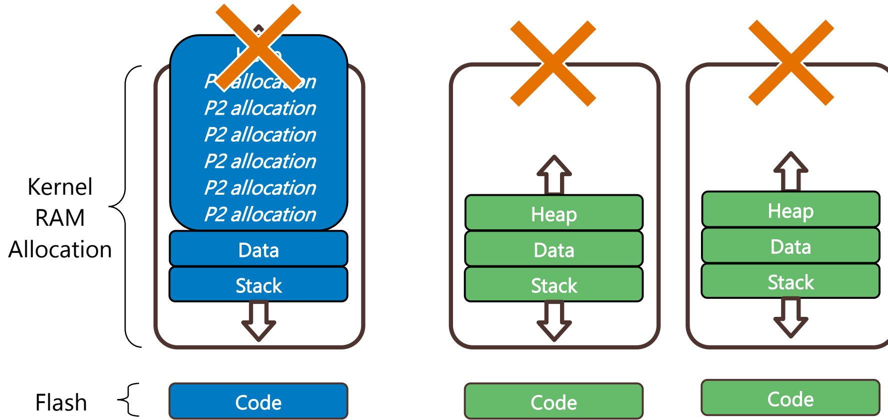
However, what happens when the app requests services from the kernel? For example, if process #1 wants the kernel to schedule a timer for it, the kernel must allocate state for that.



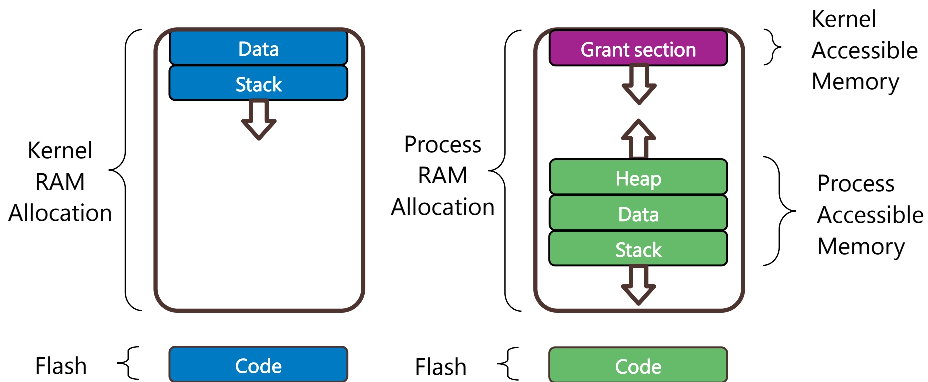
Allocating state for app requests in kernel memory works for a little while...



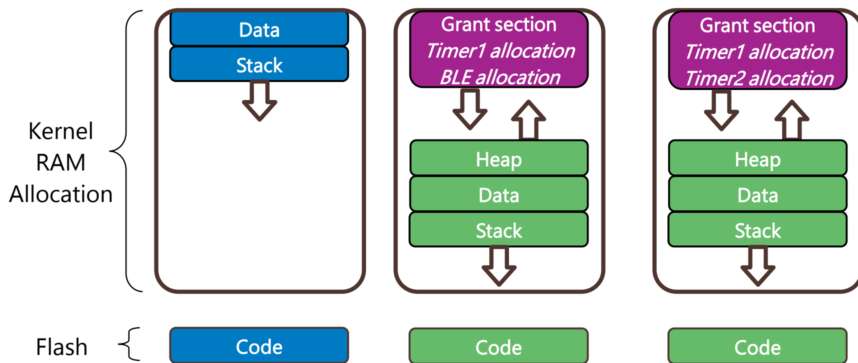
But what happens when the kernel no longer has room for all of the requests? What is the kernel's recourse? Reboot and hope it doesn't happen again?



To address this, Tock introduces grants, which are regions of application memory space that the kernel uses to store the app-specific kernel state. The grant region is not accessible to the process, and the kernel does not support dynamic memory allocation.

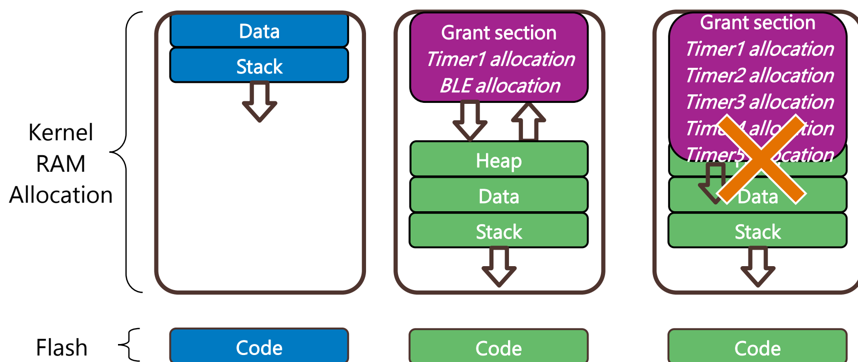


As processes want to use kernel resources, the state for those resources is allocated from the grant region in each process.



43

If one process requests too many resources, its grant section will exhaust its available memory, and the kernel can safely terminate just that process.



44

The Rust type system and closures prevents using the grant region of terminated processes.

```
fn enter<'a, F>(&'a self, pid: ProcId, f: F) → where
   F: for<'b> FnOnce(&'b mut T)
      // Can't operate on timer data here

      timer_grant.enter(process_id, |timer| {
         // Can operate on timer data here
         if timer.expiration > cur_time {
            timer.fired = true;
         }
      });

      // timer data can't escape here
```

45

Tock

- Pushing towards a general purpose OS on embedded HW
- “Users” not statically known at compile time
- Grants enable dynamic memory without crashing the kernel

46

Real Time OS

What does “Real Time” mean?

- In computing broadly
  - Real time = the time measured by a physical clock
  - Real time operation generally means the execution keeps up with the pace of events
    - A speech or video sample of 1 second, if processed in 1 second or in less
- In real time systems
  - Task execution meets specific deadlines

Real Time Operating System

- Real Time Tasks
  - Process control in industrial plants
  - Robotics
  - Air Traffic control
  - Telecommunications
  - Weapon guidance system e.g. Guided missiles
  - Medical diagnostic and life support system
  - Automatic engine control system
  - Real time data base
  - Mars Rovers
    - Curiosity : OS – VxWorks, Processor BEA's RAD 750

Terms and definitions

- Release time (or ready time): This is the time instant at which a task(process) is ready or eligible for execution
- Schedule Time: This is the time instant when a task gets its chance to execute
- Completion time: This is the time instant when task completes its execution
- Deadline: This is the instant of time by which the execution of task should be completed
- Runtime: The time taken without interruption to complete the task, after the task is released

Soft and Hard Real Time tasks

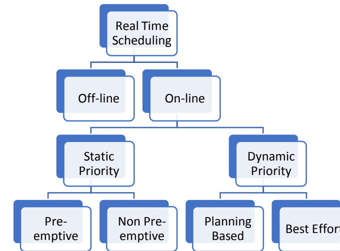
- Hard real time task:
  - Task must complete before or at deadline.
  - Value of completing the task after deadline is zero.
- Software real time
  - Missing deadline incurs a penalty
  - Penalty increases as tardiness increases

Scheduling concerns

- Scheduling constraints for real time operating systems:
  - Number of tasks
  - Resource Requirements
  - Release Time
  - Execution time
  - Deadlines
  - Period
- Simplifying assumption: execution time and deadline known, tasks execute at fixed intervals
- Goal: determine a schedule where all tasks meet their deadlines

Off Line Scheduling (Pre runtime scheduling)

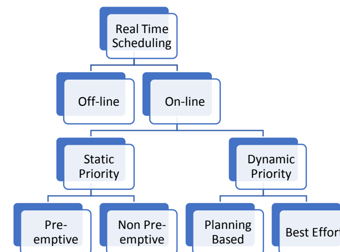
- Generate scheduling information prior to system execution (Deterministic System Model)
- This scheduling is based on:
  - Release time
  - Deadlines
  - Execution
- Disadvantage: Inflexibility, if any parameter changes, the policy will have to be recomputed



56

On-Line Scheduling

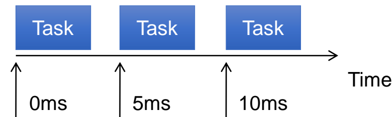
- Number and types of tasks, associated parameters are not known in advance.
- Scheduling must accommodate dynamic changes
- Online Scheduling are of two types:
  - Static Priority
  - Dynamic Priority



57

Real-Time Scheduling

Real-time tasks execute repeatedly (usually are *periodic*) under some time constraint



E.g., a task is *released* to execute every 5 msec, and each invocation has a *deadline* of 5 msec

Separate priority range from the nice priorities for CFS:

- Priorities are from 1 (low) to 99 (high), highest ones need root

Conflicts with traditional kernel scheduling

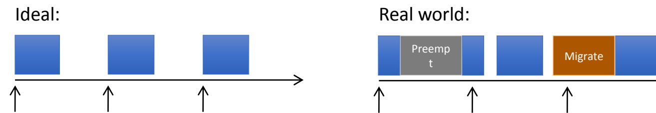
- Deadlines vs. fairness

- For example, if a user accessed the kernel: “can you guarantee my task will run for 1 second at every 5 second interval?”

- Challenges:
  - Linux uses proportional sharing – so the answer is highly dependent on other system activity
  - What if another process boosts its priority?
  - What if another process is starved?

Real-Time OS Support

Goal is to achieve predictable execution:



Other sources of uncertainty (and solutions):
- Interrupts (can mask some interrupts)
- Migrations (can pin tasks to cores)
- OS latency, jitter, and kernel non-preemptibility (real-time scheduling)

60

Five real-time scheduling classes

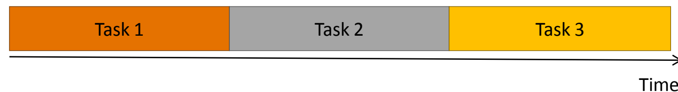
- First-in, First-out scheduling
- Round robin scheduling
- Preemptive fixed priority scheduling
- Most frequent first
- Earliest deadline first

61

FIFO Scheduling

First-in, First-out scheduling

- The first enqueued task of highest priority executes to completion
- A task will only relinquish a processor when it completes, yields, or blocks



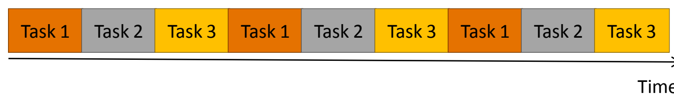
- Only a higher priority `SCHED_FIFO` or `SCHED_RR` task can preempt a `SCHED_FIFO` task
  - all others will be starved as it runs

62

Round Robin Scheduling

Round-robin scheduling
Same as `SCHED_FIFO` but with timeslices

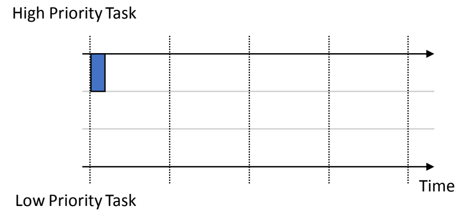
Among tasks of equal priority:
- Rotate through all tasks
- Each task gets a fixed time slice



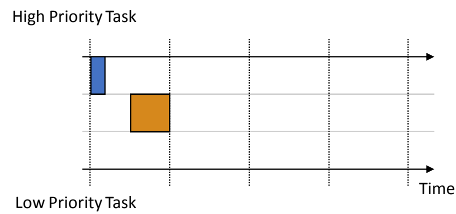
- Only a higher priority `SCHED_FIFO` or `SCHED_RR` task can preempt a `SCHED_FIFO`
- Tasks of equal priority preempt each other after timeslice expiration

63

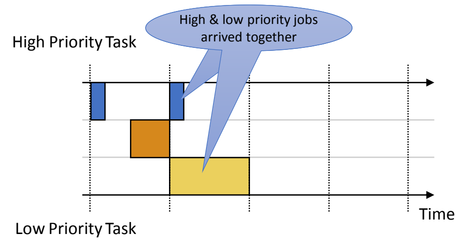
Preemptive Fixed Priority Scheduling



Preemptive Fixed Priority Scheduling

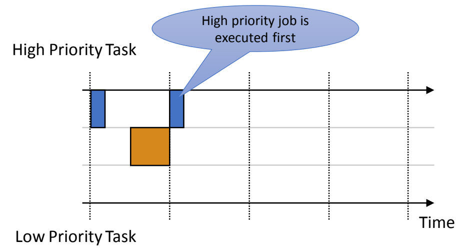


Preemptive Fixed Priority Scheduling



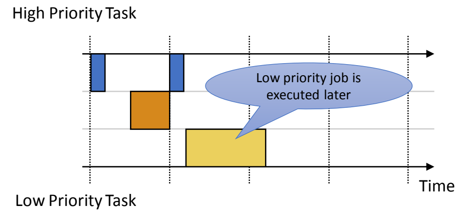
66

Preemptive Fixed Priority Scheduling

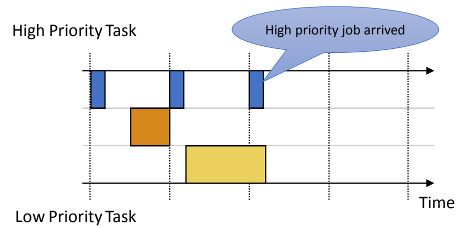


67

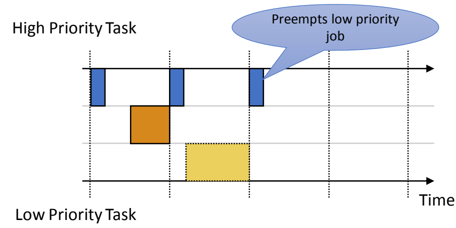
Preemptive Fixed Priority Scheduling



Preemptive Fixed Priority Scheduling

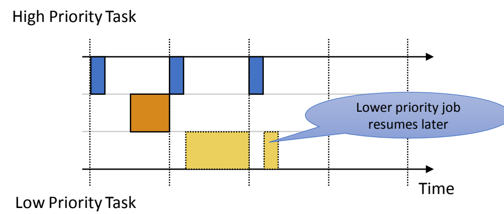


Preemptive Fixed Priority Scheduling



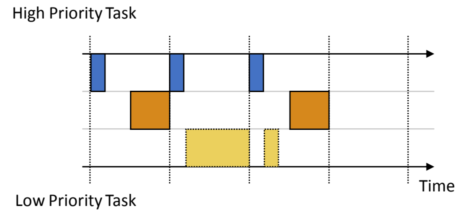
70

Preemptive Fixed Priority Scheduling



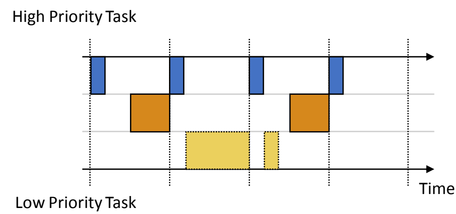
71

Preemptive Fixed Priority Scheduling



72

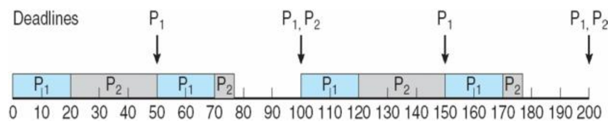
Preemptive Fixed Priority Scheduling



73

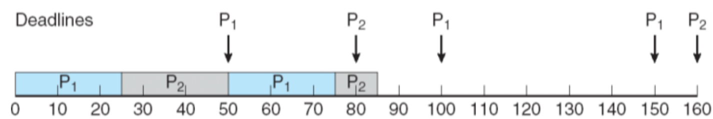
Rate Montonic Scheduling

- A priority is assigned based on the inverse of its period
- Shorter periods = higher priority
- Longer periods = lower priority
- P_1 is assigned a higher priority than P_2.



74

Missed Deadlines with Rate Monotonic Scheduling



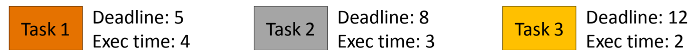
75

EDF Scheduling

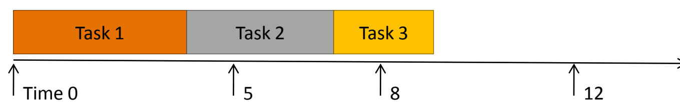
Earliest Deadline First (EDF) scheduling

- Simple, yet effective

- Whichever task has next deadline gets to run



- Theory exists to analyze such systems



Earliest Deadline First

- EDF is also optimal
  - If the utility is <= 1 then EDF will create a schedule
  - Can transform any feasible schedule into an EDF one
- But this is based on assumptions
  - Perfect periodic tasks, zero overhead, independence

EDF Seem Ideal

- Easy to implement, optimal, …
- But only in an ideal world
  - If the workload is briefly non-schedulable
    - Then both fail dramatically and unpredictably
  - Choices are based on deadlines, not importance
    - Work hard to display MPG rather than firing airbag
- Work only with idealized situations
  - We need to relax the simplifying conditions

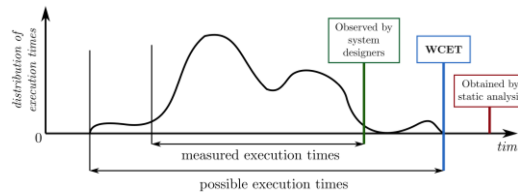
78

Mixed Criticality

90

Real time systems: all about guarantees

- A lot of real time scheduling is based on assumptions
  - Periodic tasks
  - Known priorities
  - Known runtimes
- What happens when we don't know how long something will execute?
  - Runtime can vary
  - Still want to meet deadlines
- Mixed criticality embraces this uncertainty



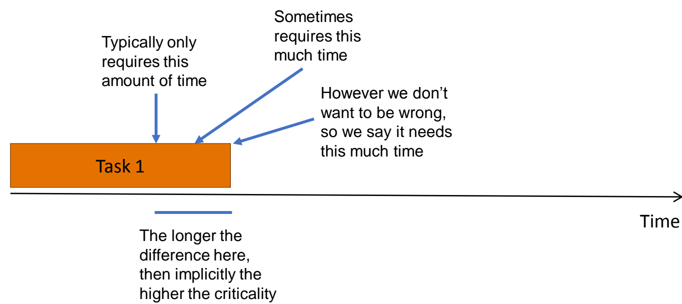
91

Mixed Criticality - The Vestal Model

- Introduces the notion that confidence in a software task's Worst Case Execution Time (WCET) C is proportional to its criticality
  - Higher criticality. Higher analysis effort. More pessimistic WCET.
  - CA ≥ CB ≥ CC ≥ CD ≥ CE
- Focusing on a dual criticality system…academic works and models that build off Vestal's seminal work essentially treat each task as having a WCET for each criticality.
  - High DAL tasks => C_{HI} and C_{LO}
  - Low DAL tasks => C_{LO}
- The question is how to utilise the spare execution time – [CHI - CLO]

92

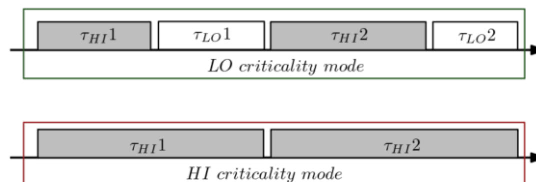
Mixed Criticality intuition



93

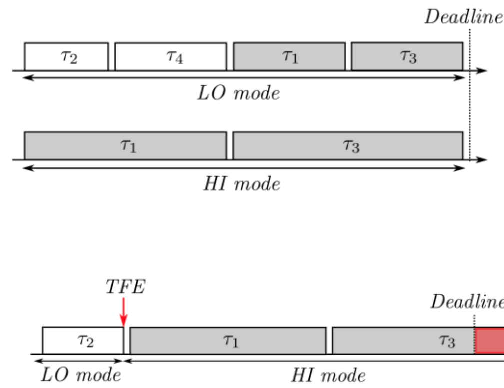
LO and HI criticality modes

- Ci(LO): Max. observed execution time (system designers).
- Ci(HI): Upper-bounded execution time (static analysis).



94

Scheduling with mode changes



Mixed criticality systems

- Integrating real-world issues into real time systems
- Interesting use of uncertainty to guide design decisions

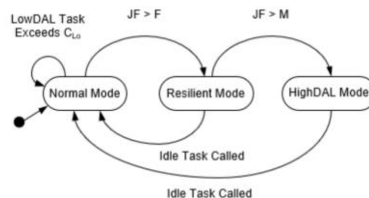
Robust Mixed-Criticality

- Normal mode
  - F hi-criticality tasks can exceed C_{LO}
- Resilient mode
  - Up to M tasks can exceed C_{LO}
  - Each robust task can skip up to S jobs
- High-criticality mode
  - Low-criticality tasks are not released
- On an idle tick the counters for jobs skipped (JF) are reset
- If C_{Hi} is exceeded then there is a power cycle

97

Robust Mixed Criticality

- A ***robust task*** is one that can safely drop a non-started job
- A ***fault*** is measured when one task overruns its C_{LO}
  - JF Records the number of Job Failures
- An ***error*** is registered when one or many tasks fail to comply with their timing requirements
- A ***resilient system*** is one which employs graceful degradation (*if necessary through the control of **robust tasks***) in order to cope with one or many ***faults***, thus aiming to avoid ***errors***.



98