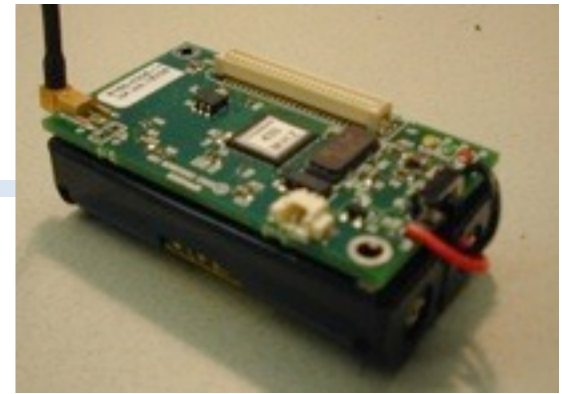
TinyOS and nesC

- TinyOS: OS for wireless sensor networks.
- nesC: programming language for TinyOS.

Mica2 Mote



- Processor
  - Microcontroller: 7.4 MHz, 8 bit
  - Memory: 4KB data, 128 KB program
- Radio
  - Max 38.4 Kbps
- Sensors
  - Light, temperature, acceleration, acoustic, magnetic…
- Power
  - <1 week on two AA batteries in active mode
  - >1 year battery life on sleep modes!

Hardware Constraints

Severe constraints on power, size, and cost →

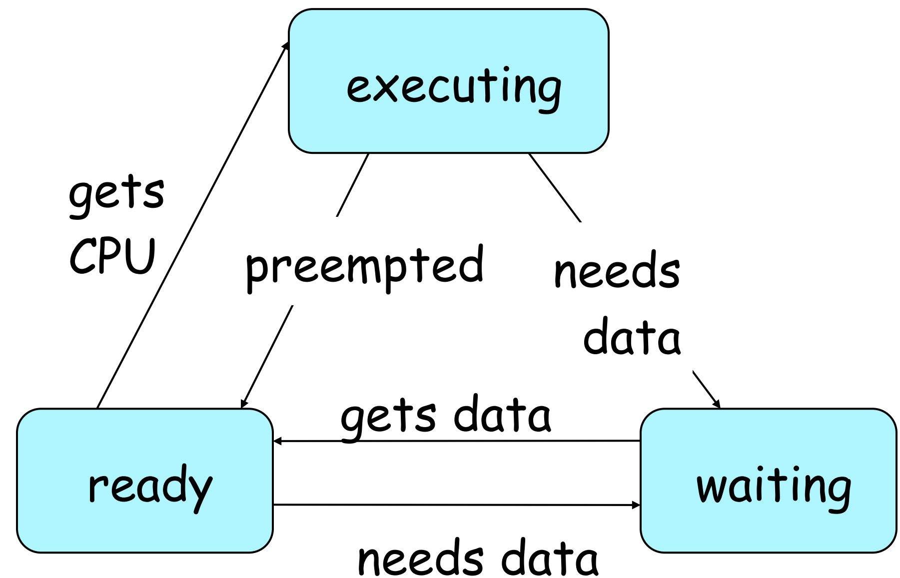
- slow microprocessor
- low-bandwidth radio
- limited memory
- limited hardware parallelism → CPU hit by many interrupts!
- manage sleep modes in hardware components

Software Challenges

- Small memory footprint
- Efficiency - power and processing
- Concurrency-intensive operations
- Diversity in applications & platform → efficient modularity
  - Support evolution of hardware and software

Traditional OS

- Multi-threaded
- Preemptive scheduling
- Threads:
  - ready to run;
  - executing on the CPU;
  - waiting for data.

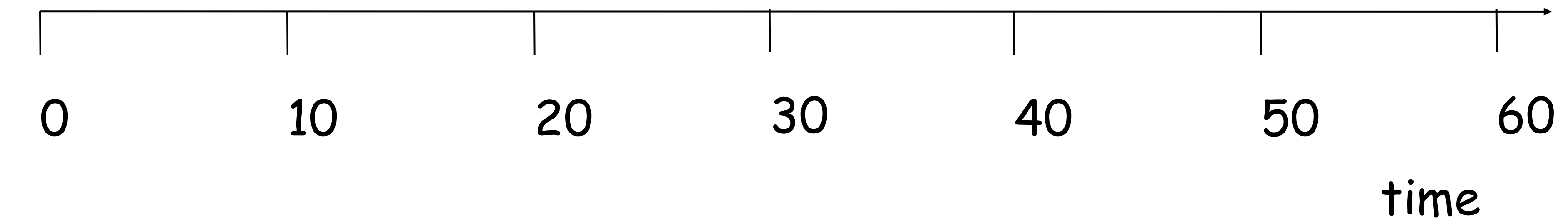


Pros and Cons of Traditional OS

- Multi-threaded + preemptive scheduling
  - Preempted threads waste memory
  - Context switch overhead
- I/O
  - Blocking I/O: waste memory on blocked threads
  - Polling (busy-wait): waste CPU cycles and power

Example: Preemptive Priority Scheduling

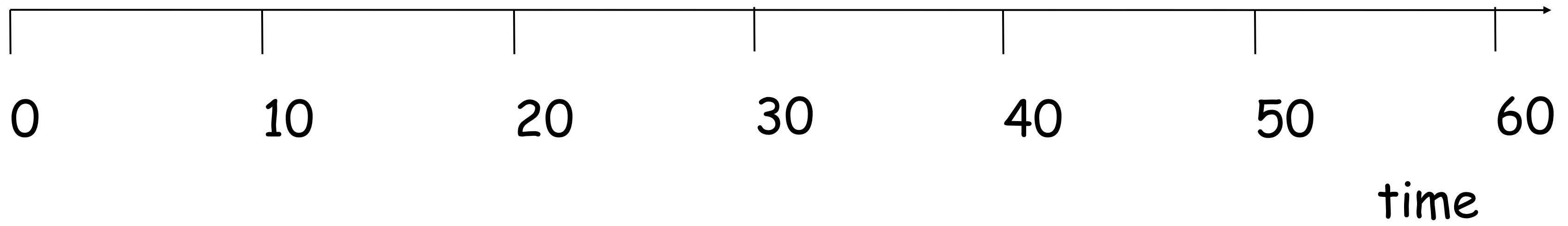
- Each process has a fixed priority (1 highest);
- P_1: priority 1; P_2: priority 2; P_3: priority 3.



Example: Preemptive Priority Scheduling

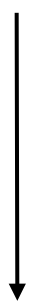
- Each process has a fixed priority (1 highest);
- P_1: priority 1; P_2: priority 2; P_3: priority 3.

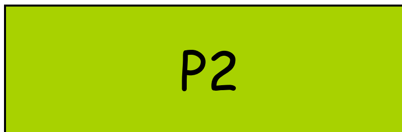
P_2 released

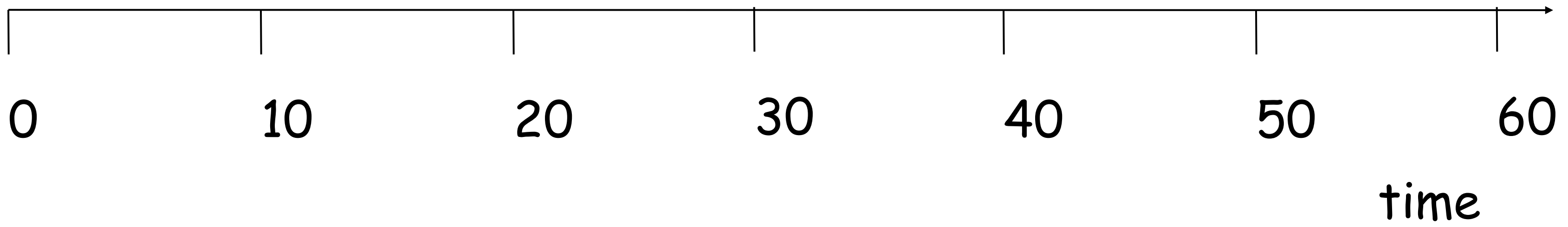


Example: Preemptive Priority Scheduling

- Each process has a fixed priority (1 highest);
- P_1: priority 1; P_2: priority 2; P_3: priority 3.



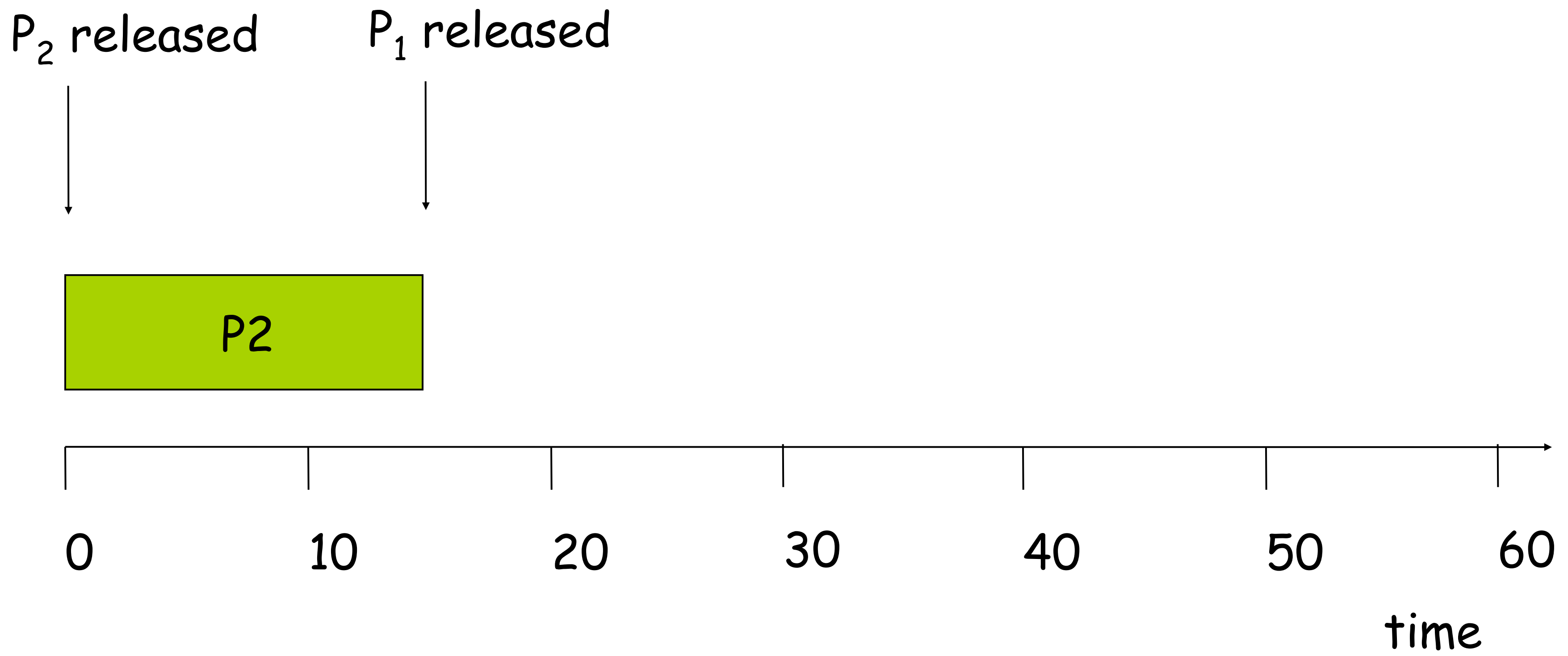




P2 released

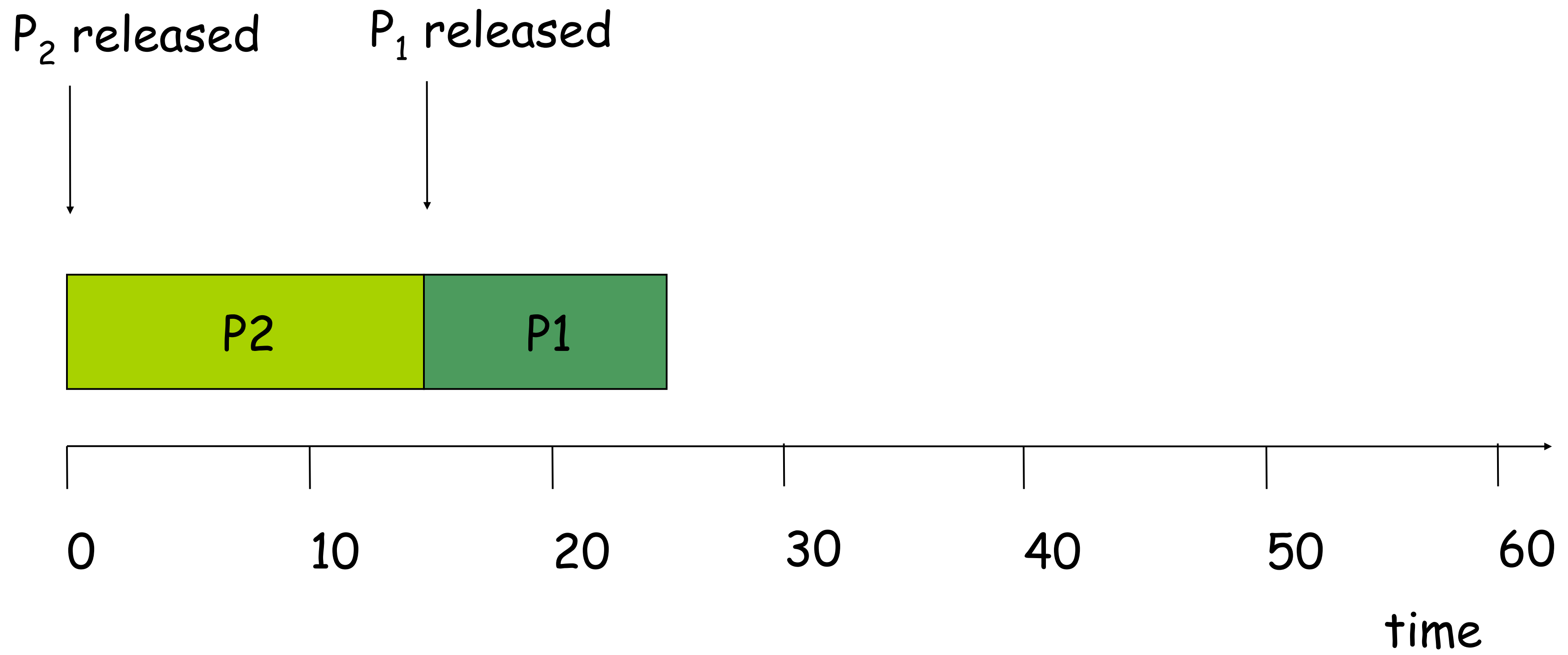
Example: Preemptive Priority Scheduling

- Each process has a fixed priority (1 highest);
- P_1: priority 1; P_2: priority 2; P_3: priority 3.



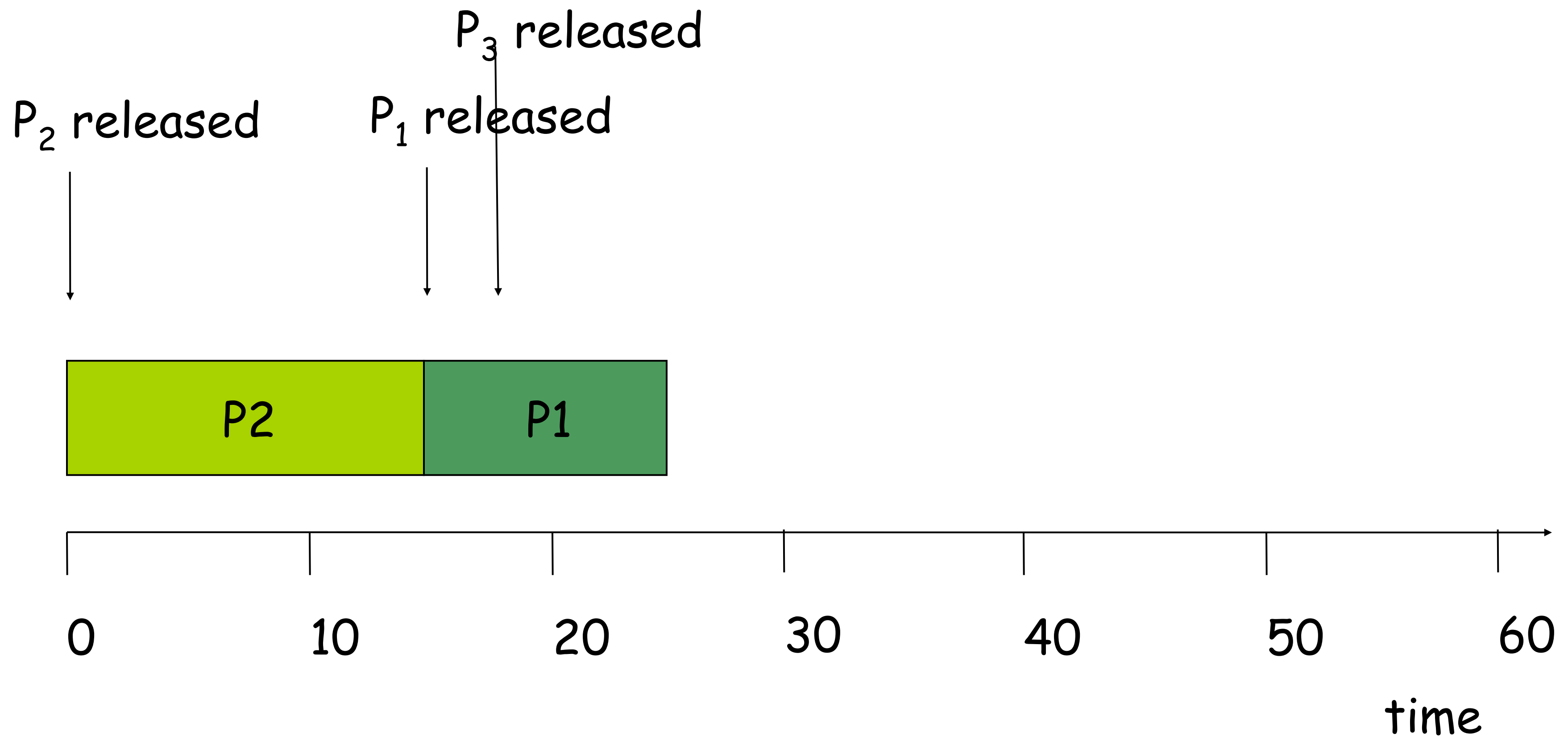
Example: Preemptive Priority Scheduling

- Each process has a fixed priority (1 highest);
- P_1: priority 1; P_2: priority 2; P_3: priority 3.



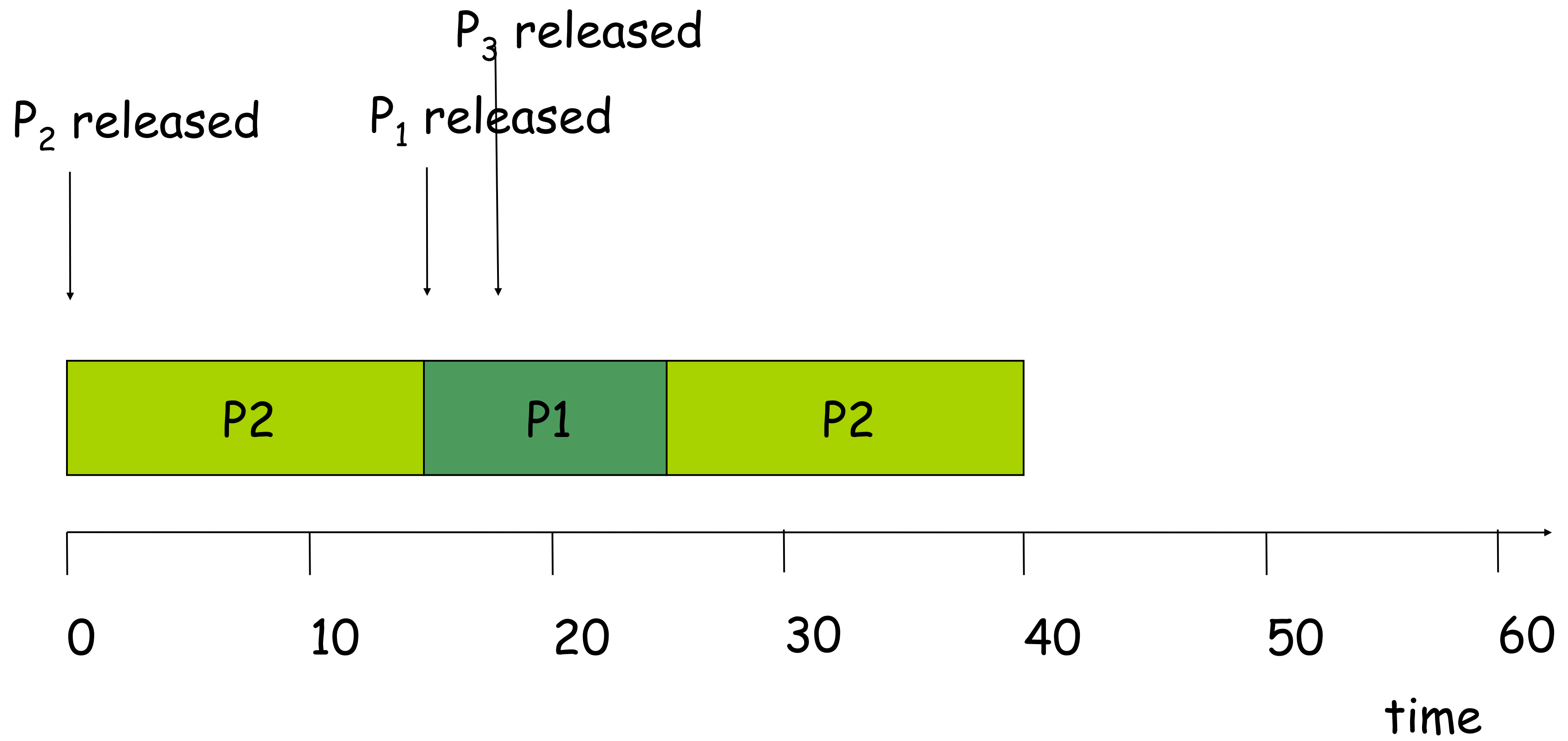
Example: Preemptive Priority Scheduling

- Each process has a fixed priority (1 highest);
- P_1: priority 1; P_2: priority 2; P_3: priority 3.



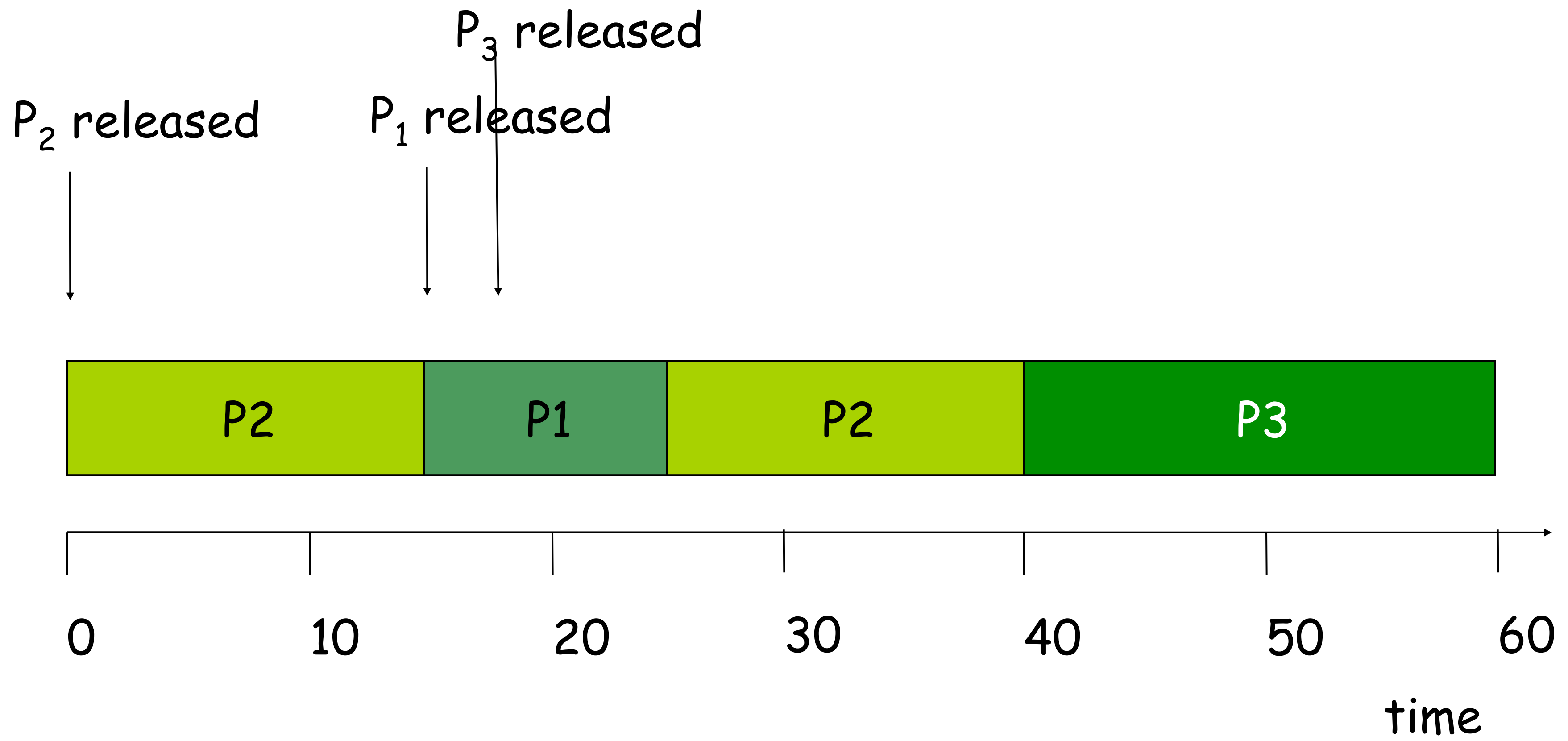
Example: Preemptive Priority Scheduling

- Each process has a fixed priority (1 highest);
- P_1: priority 1; P_2: priority 2; P_3: priority 3.

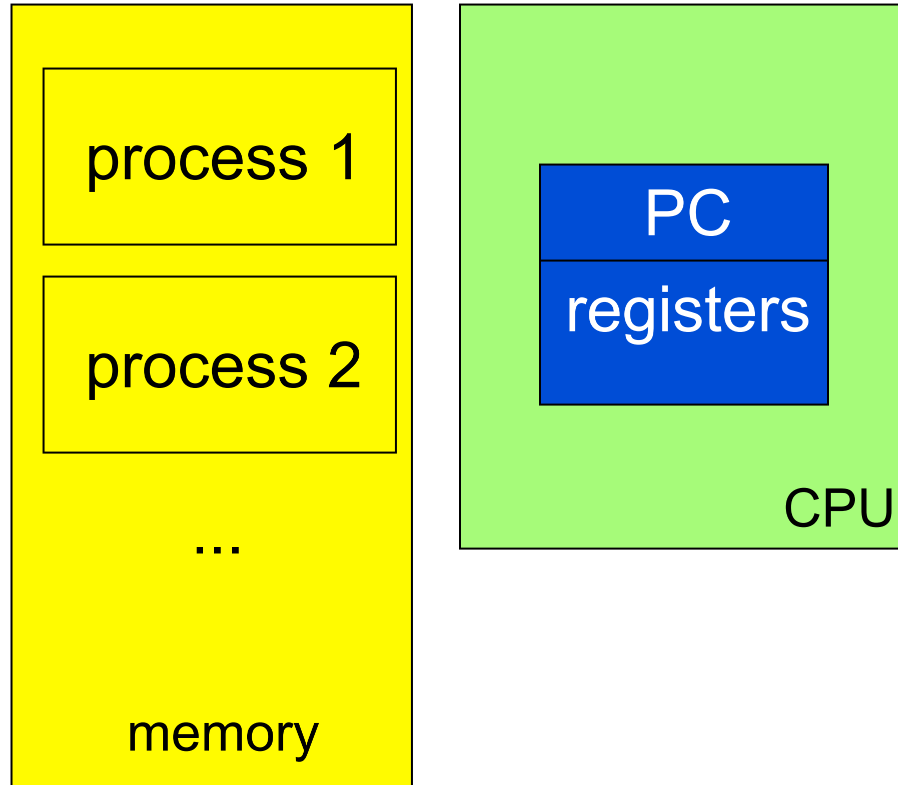


Example: Preemptive Priority Scheduling

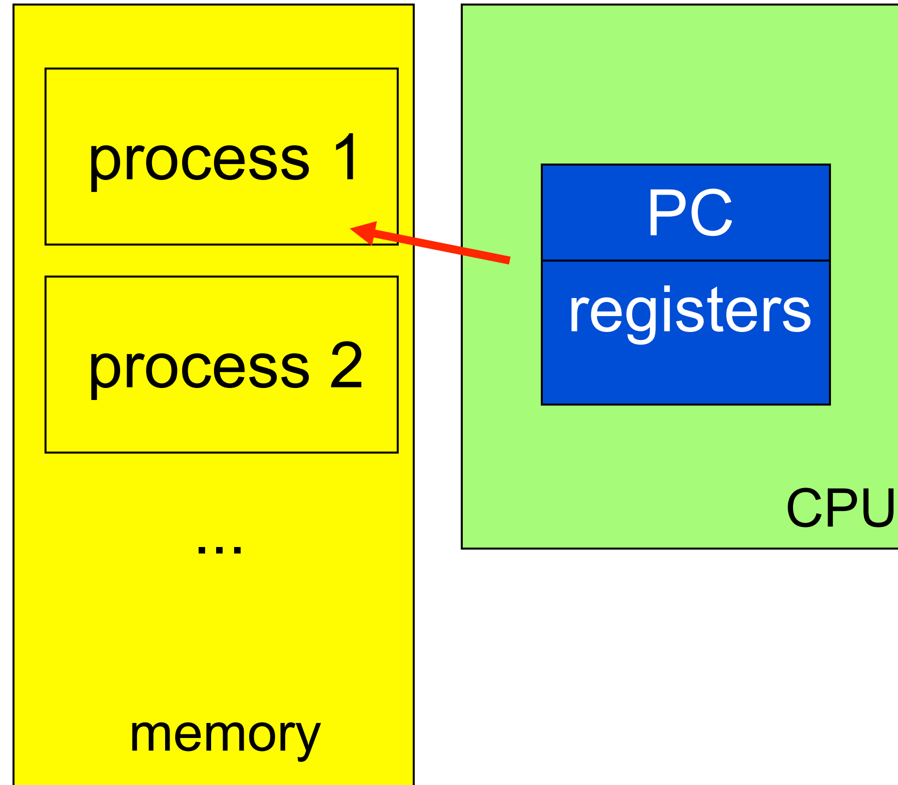
- Each process has a fixed priority (1 highest);
- P_1: priority 1; P_2: priority 2; P_3: priority 3.



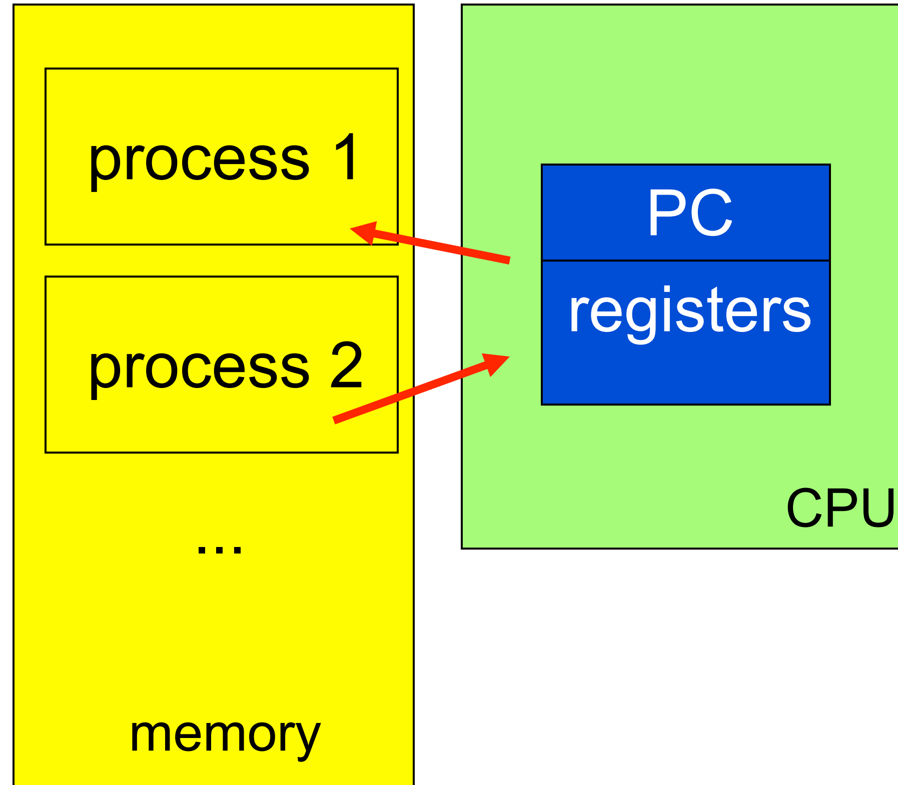
Context Switch



Context Switch



Context Switch



Existing Embedded OS

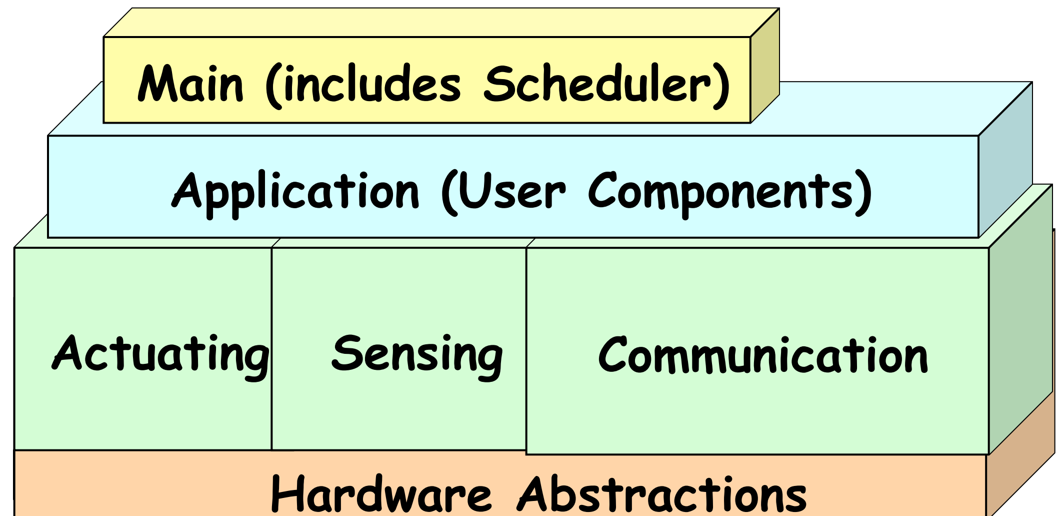
Name	Code Size	Target CPU
pOSEK	2K	Microcontrollers
pSOSystem		PII->ARM Thumb
VxWorks	286K	Pentium -> Strong ARM
QNX Nutrino	>100K	Pentium II -> NEC
QNX RealTime	100K	Pentium II -> SH4
OS-9		Pentium -> SH4
Chorus OS	10K	Pentium -> Strong ARM
ARIEL	19K	SH2, ARM Thumb
Creem	560 bytes	ATMEL 8051

- QNX context switch = 2400 cycles on x86
- pOSEK context switch > 40 µs
- Creem -> no preemption

System architecture directions for network sensors, J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, K. Pister. ASPLOS 2000.

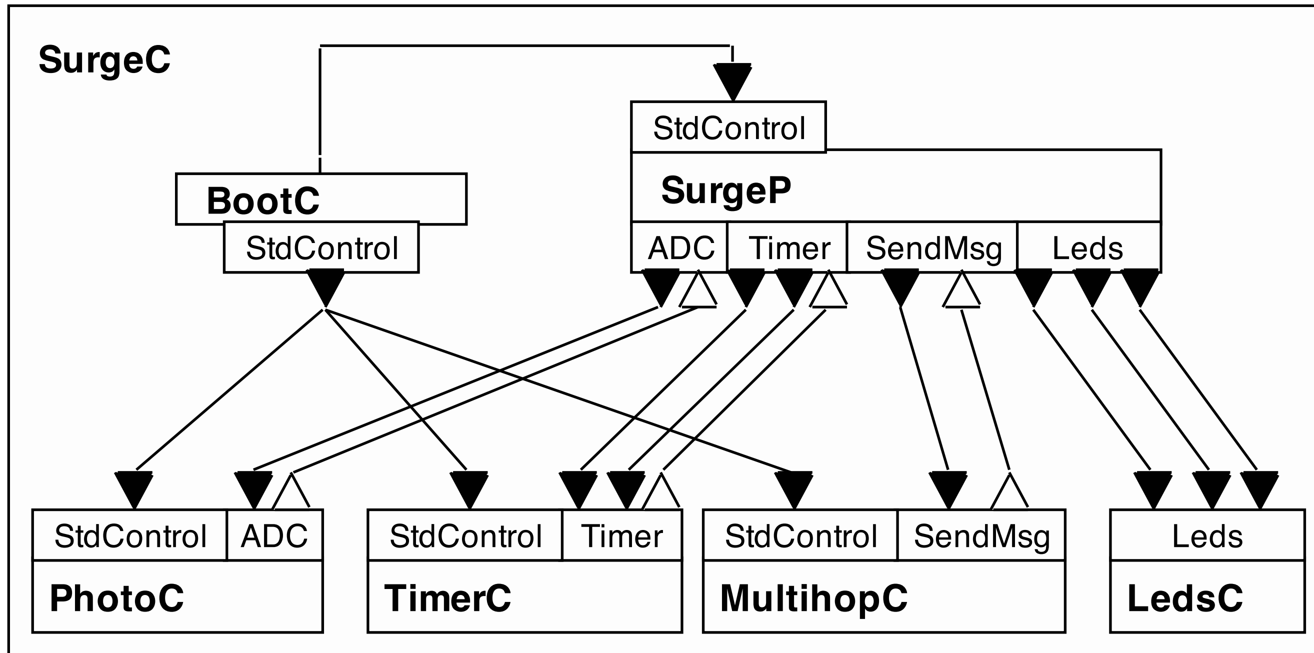
TinyOS Solutions

- Efficient modularity
  - Application = scheduler + graph of components
  - Compiled into one executable
  - Only needed components are complied/loaded
- Concurrency: event-driven architecture



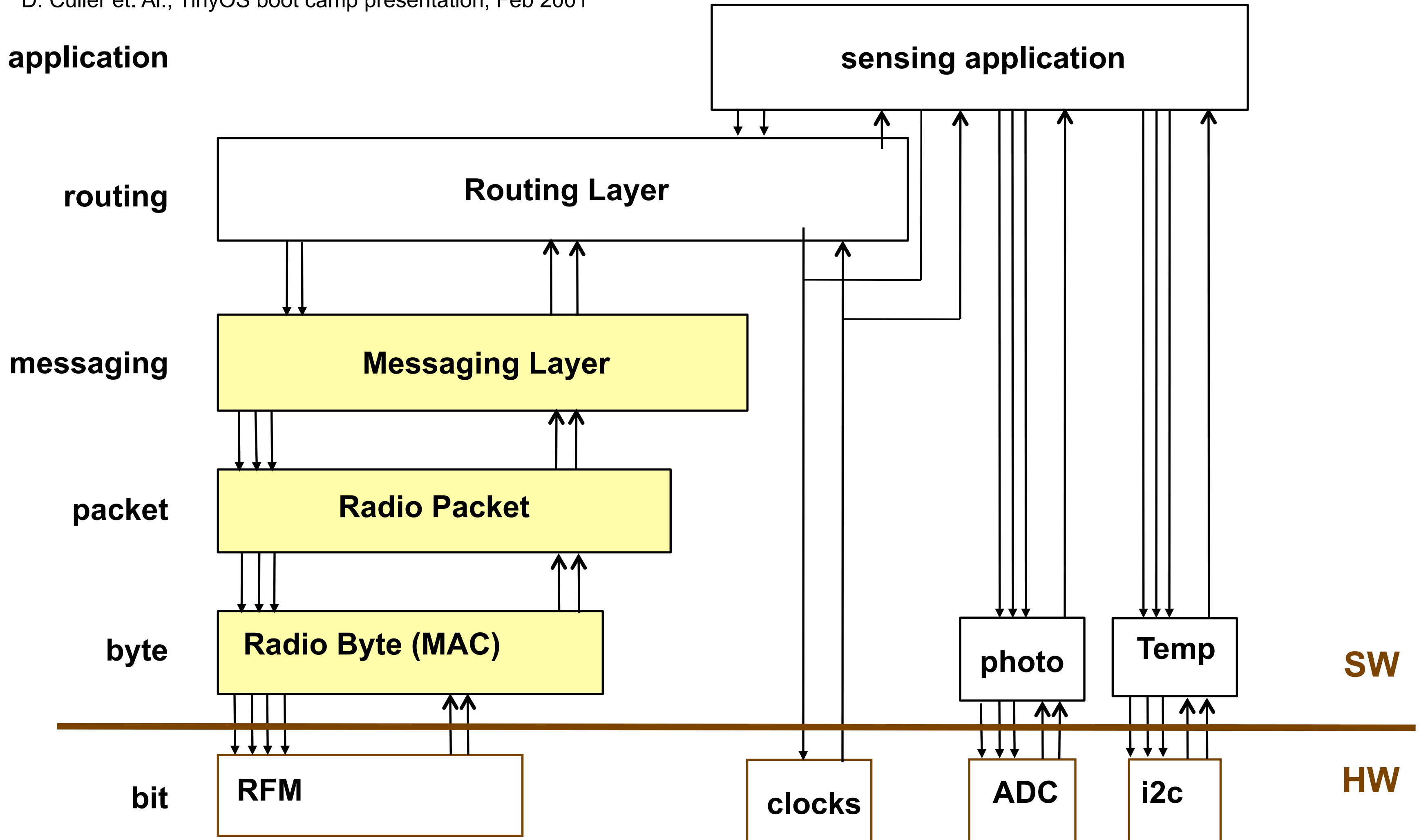
Modified from D. Culler et. Al., TinyOS boot camp presentation, Feb 2001

Example: Surge



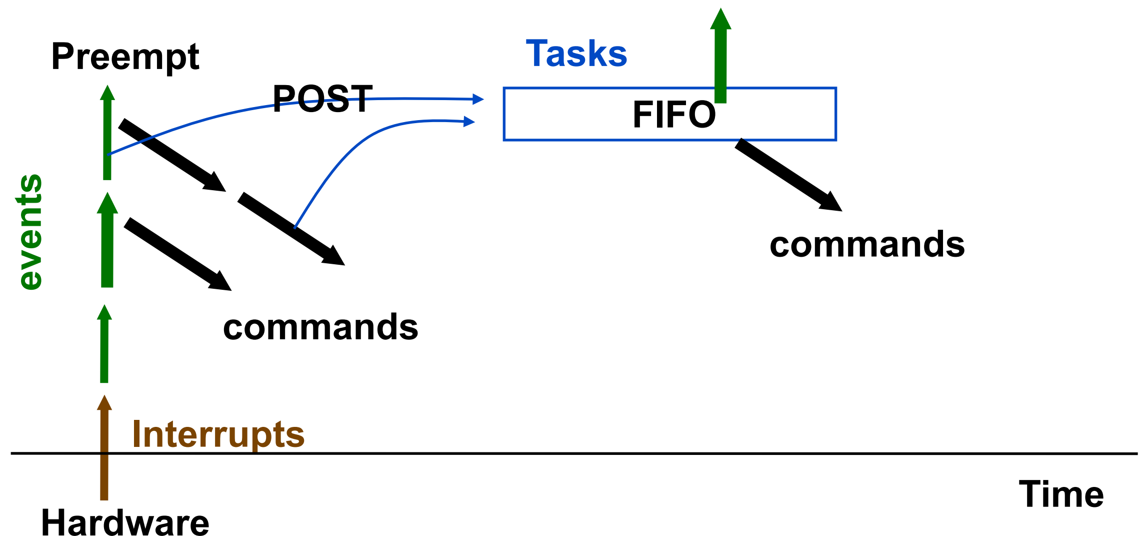
Typical Application

D. Culler et. Al., TinyOS boot camp presentation, Feb 2001



Two-level Scheduling

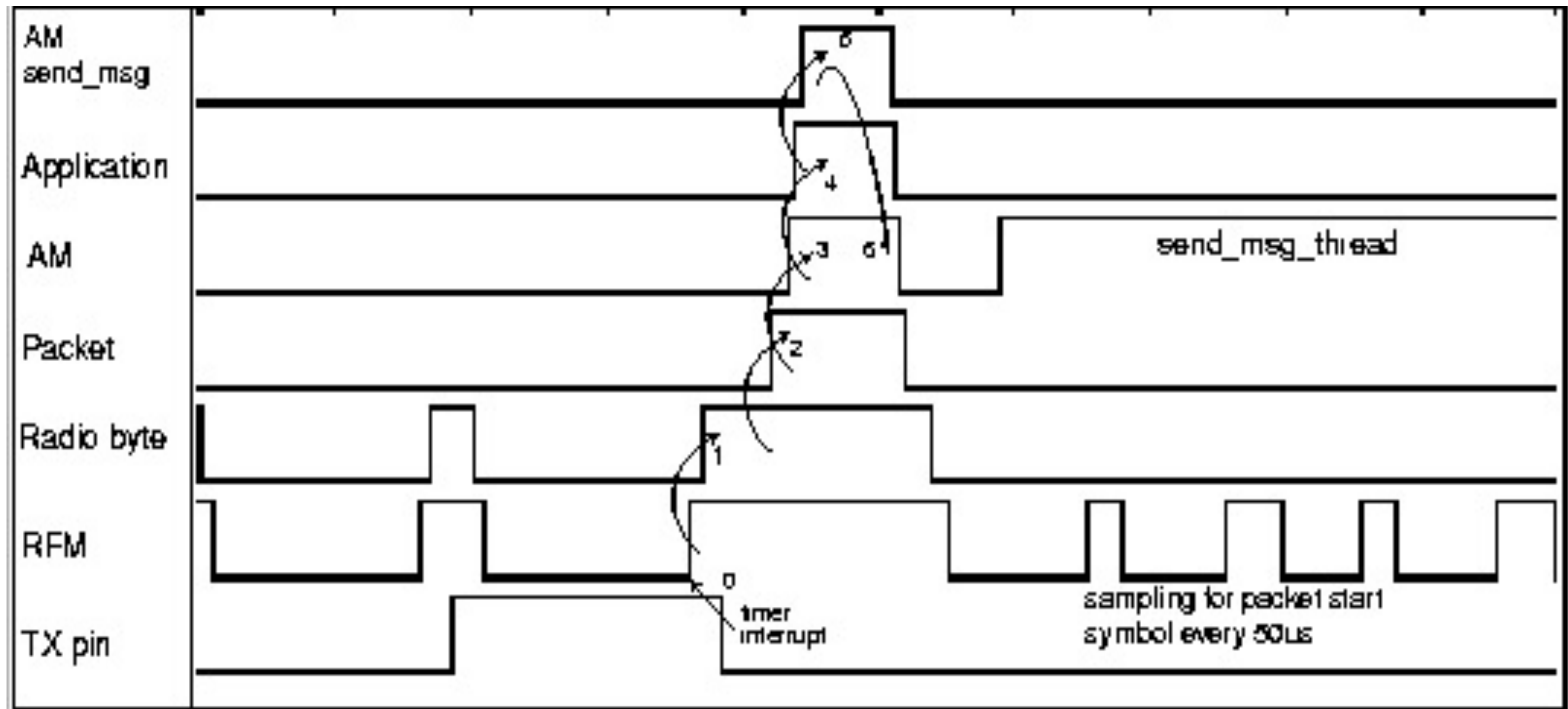
- Events handle interrupts
  - Interrupts trigger lowest level events
  - Events can signal events, call commands, or post tasks
- Tasks perform deferred computations
- Interrupts preempt tasks and interrupts



Multiple Data Flows

- Respond quickly: sequence of event/command through the component graph.
  - Immediate execution of function calls
  - e.g., get bit out of radio hw before it gets lost.
- Post tasks for deferred computations.
  - e.g., encoding.
- Events preempt tasks to handle new interrupts.

Sending a Message



Timing diagram of event propagation

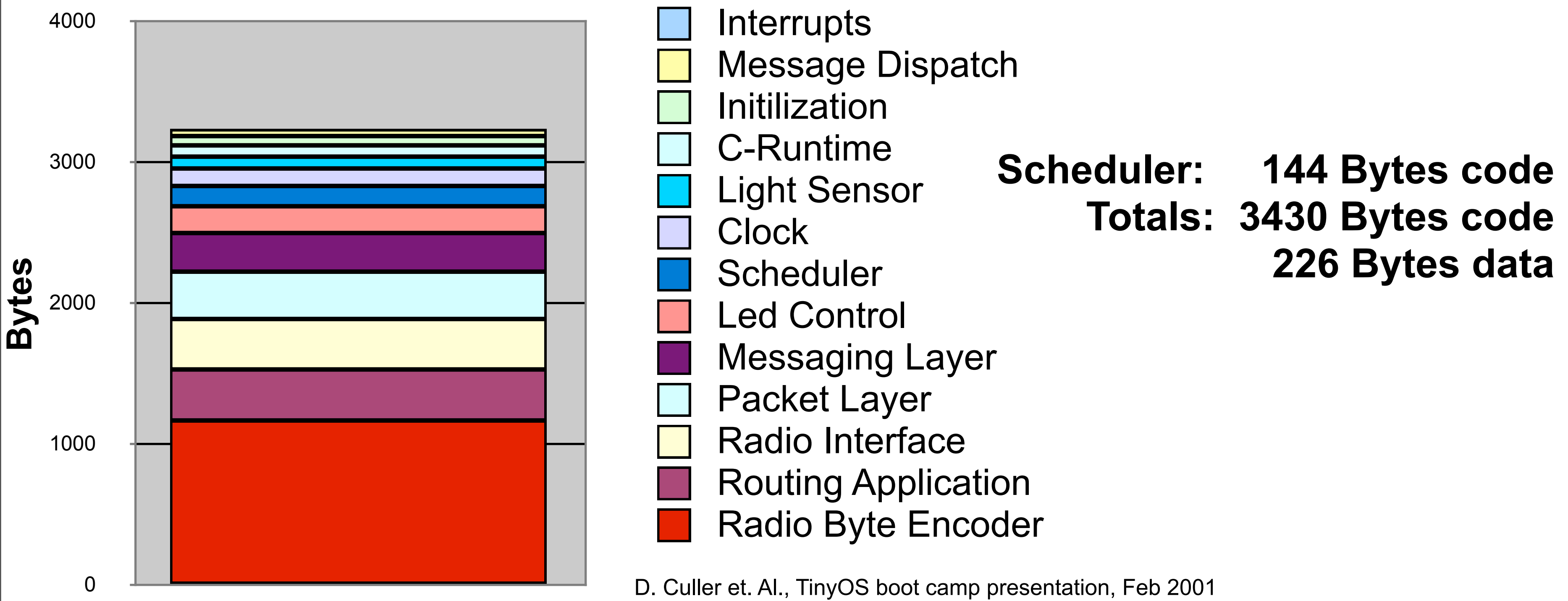
(step 0-6 takes about 95 microseconds total)

Scheduling

- Interrupts preempt tasks
  - Respond quickly
  - Event/command implemented as function calls
- Task cannot preempt tasks
  - Reduce context switch → efficiency
  - Single stack → low memory footprint
  - TinyOS 2 supports pluggable task scheduler (default: FIFO).
- Scheduler puts processor to sleep when
  - no event/command is running
  - task queue is empty

Space Breakdown…

Code size for ad hoc networking application



Power Breakdown…

	Active	Idle	Sleep
CPU	5 mA	2 mA	5 µA
Radio	7 mA (TX)	4.5 mA (RX)	5 µA
EE-Prom	3 mA	0	0
LED's	4 mA	0	0
Photo Diode	200 µA	0	0
Temperature	200 µA	0	0

Panasonic
CR2354
560 mAh

- Lithium Battery runs for 35 hours at peak load and years at minimum load!
  - That's three orders of magnitude difference!
- A one byte transmission uses the same energy as approx 11000 cycles of computation.

Time Breakdown…

Components	Packet reception work breakdown	CPU Utilization	Energy (nj/Bit)
AM	0.05%	0.20%	0.33
Packet	1.12%	0.51%	7.58
Ratio handler	26.87%	12.16%	182.38
Radio decode thread	5.48%	2.48%	37.2
RFM	66.48%	30.08%	451.17
Radio Reception	-	-	1350
Idle	-	54.75%	-
Total	100.00%	100.00%	2028.66

- 50 cycle task overhead (6 byte copies)
- 10 cycle event overhead (1.25 byte copies)

Time Breakdown…

Components	Packet reception work breakdown	CPU Utilization	Energy (nj/Bit)
AM	0.05%	0.20%	0.33
Packet	1.12%	0.51%	7.58
Ratio handler	26.87%	12.16%	182.38
Radio decode thread	5.48%	2.48%	37.2
RFM	66.48%	30.08%	451.17
Radio Reception	-	-	1350
Idle	-	54.75%	-
Total	100.00%	100.00%	2028.66

- 50 cycle task overhead (6 byte copies)
- 10 cycle event overhead (1.25 byte copies)

Time Breakdown…

Components	Packet reception work breakdown	CPU Utilization	Energy (nj/Bit)
AM	0.05%	0.20%	0.33
Packet	1.12%	0.51%	7.58
Ratio handler	26.87%	12.16%	182.38
Radio decode thread	5.48%	2.48%	37.2
RFM	66.48%	30.08%	451.17
Radio Reception	-	-	1350
Idle	-	54.75%	-
Total	100.00%	100.00%	2028.66

- 50 cycle task overhead (6 byte copies)
- 10 cycle event overhead (1.25 byte copies)

Time Breakdown…

Components	Packet reception work breakdown	CPU Utilization	Energy (nj/Bit)
AM	0.05%	0.20%	0.33
Packet	1.12%	0.51%	7.58
Ratio handler	26.87%	12.16%	182.38
Radio decode thread	5.48%	2.48%	37.2
RFM	66.48%	30.08%	451.17
Radio Reception	-	-	1350
Idle	-	54.75%	-
Total	100.00%	100.00%	2028.66

- 50 cycle task overhead (6 byte copies)
- 10 cycle event overhead (1.25 byte copies)

Time Breakdown…

Components	Packet reception work breakdown	CPU Utilization	Energy (nj/Bit)
AM	0.05%	0.20%	0.33
Packet	1.12%	0.51%	7.58
Ratio handler	26.87%	12.16%	182.38
Radio decode thread	5.48%	2.48%	37.2
RFM	66.48%	30.08%	451.17
Radio Reception	-	-	1350
Idle	-	54.75%	-
Total	100.00%	100.00%	2028.66

- 50 cycle task overhead (6 byte copies)
- 10 cycle event overhead (1.25 byte copies)

Time Breakdown…

Components	Packet reception work breakdown	CPU Utilization	Energy (nj/Bit)
AM	0.05%	0.20%	0.33
Packet	1.12%	0.51%	7.58
Ratio handler	26.87%	12.16%	182.38
Radio decode thread	5.48%	2.48%	37.2
RFM	66.48%	30.08%	451.17
Radio Reception	-	-	1350
Idle	-	54.75%	-
Total	100.00%	100.00%	2028.66

- 50 cycle task overhead (6 byte copies)
- 10 cycle event overhead (1.25 byte copies)

Time Breakdown…

Components	Packet reception work breakdown	CPU Utilization	Energy (nj/Bit)
AM	0.05%	0.20%	0.33
Packet	1.12%	0.51%	7.58
Ratio handler	26.87%	12.16%	182.38
Radio decode thread	5.48%	2.48%	37.2
RFM	66.48%	30.08%	451.17
Radio Reception	-	-	1350
Idle	-	54.75%	-
Total	100.00%	100.00%	2028.66

- 50 cycle task overhead (6 byte copies)
- 10 cycle event overhead (1.25 byte copies)

Advantages

- Small memory footprint
  - Only needed components are complied/loaded
  - Single stack for tasks
- Power efficiency
  - Put CPU to sleep whenever the task queue is empty
  - TinyOS 2 provides power management for peripherals and microprocessors (ICEM).
- Efficient modularity
  - Event/command interfaces between components
  - Event/command implemented as function calls
- Concurrency-intensive operations
  - Event/command + tasks

Issues

- Lack preemptive real-time scheduling
  - Urgent task may wait for non-urgent ones
- Lack flexibility
  - Static linking only
  - Cannot change parts of the code dynamically
- Unfamiliar APIs
  - POSIX thread library in TinyOS 2.x mitigates the problem
- No protection barrier between applications and kernel

More

- Multi-threaded vs. event-driven architectures
  - Lack empirical comparison against existing OSes
  - A “standard” OS is more likely to be adopted by industry
  - Jury is still out…
- Alternative: Native Java Virtual Machine
  - Java programming
  - Virtual machine provides protection
  - Example: Sun SPOT

nesC

- Programming language for TinyOS and applications
- Support TinyOS components
- Whole-program analysis at compile time
  - Improve robustness: detect race conditions
  - Optimization: function inlining
- Static language
  - No function pointer
  - No malloc
  - Call graph and variable access are known at compile time

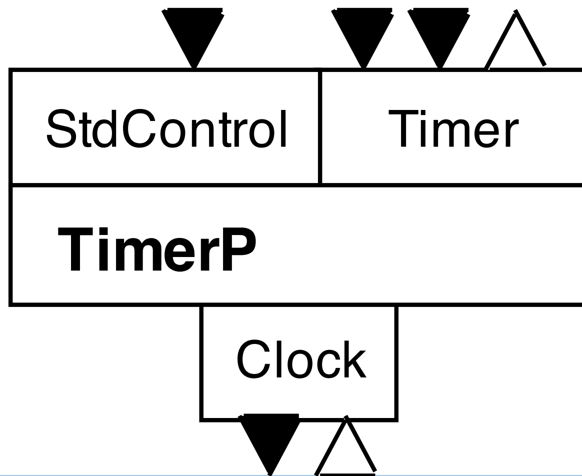
Interfaces

```
interface Clock {
    command error_t setRate(char interval, char scale);
    event error_t fire();
}

interface Send {
    command error_t send(message_t *msg, uint16_t length);
    event error_t sendDone(message_t *msg, error_t success);
}

interface ADC {
    command error_t getData();
    event error_t dataReady(uint16_t data);
}
```

- Interfaces are bi-directional
- They include both ***commands*** and ***events***
- Java interfaces are a good analogy

Modules

- Implement functionality - C-like syntax
- Provide and use sets of interfaces
  - provider of an interface implements its commands
  - users of an interface implements its events
- WARNING: modules != objects
  - each module has a single "instance"



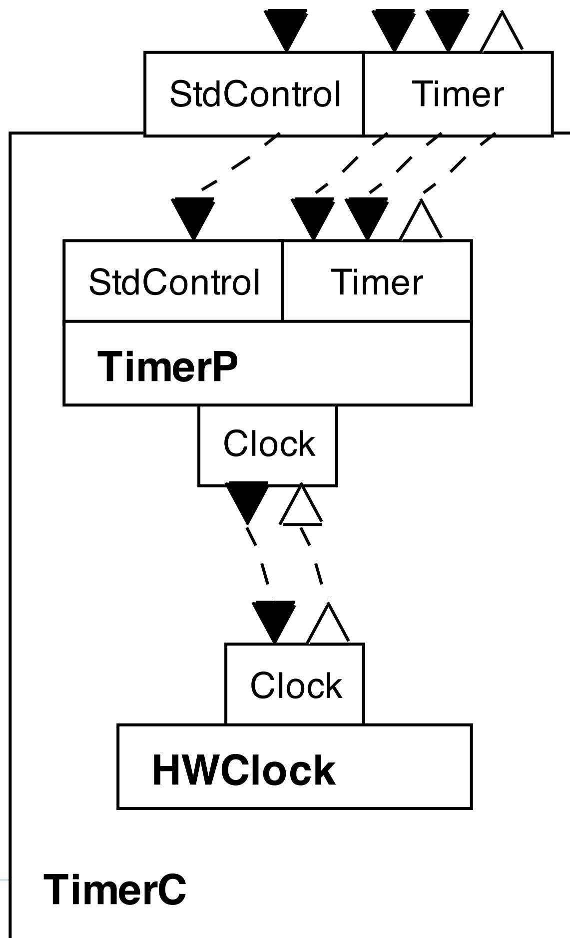

```
module TimerP {
  provides {
    interface StdControl;
    interface Timer;
  }
  uses interface Clock;
  ...
}
```

Module

```
module SurgeP {
    provides interface StdControl;
    uses interface ADC;
    uses interface Timer;
    uses interface Send;
}
implementation {
    bool busy;
    norace uint16_t sensorReading;
    async event result_t Timer.fired() {
        bool localBusy;
        atomic {
                        localBusy = busy;
                        busy = TRUE;
        }
        if (!localBusy) call ADC.getData();
        return SUCCESS;
    }
    async event result_t ADC.dataReady(uint16_t data) {
        sensorReading = data;
        post sendData();
        return SUCCESS;
    } ...
}
```

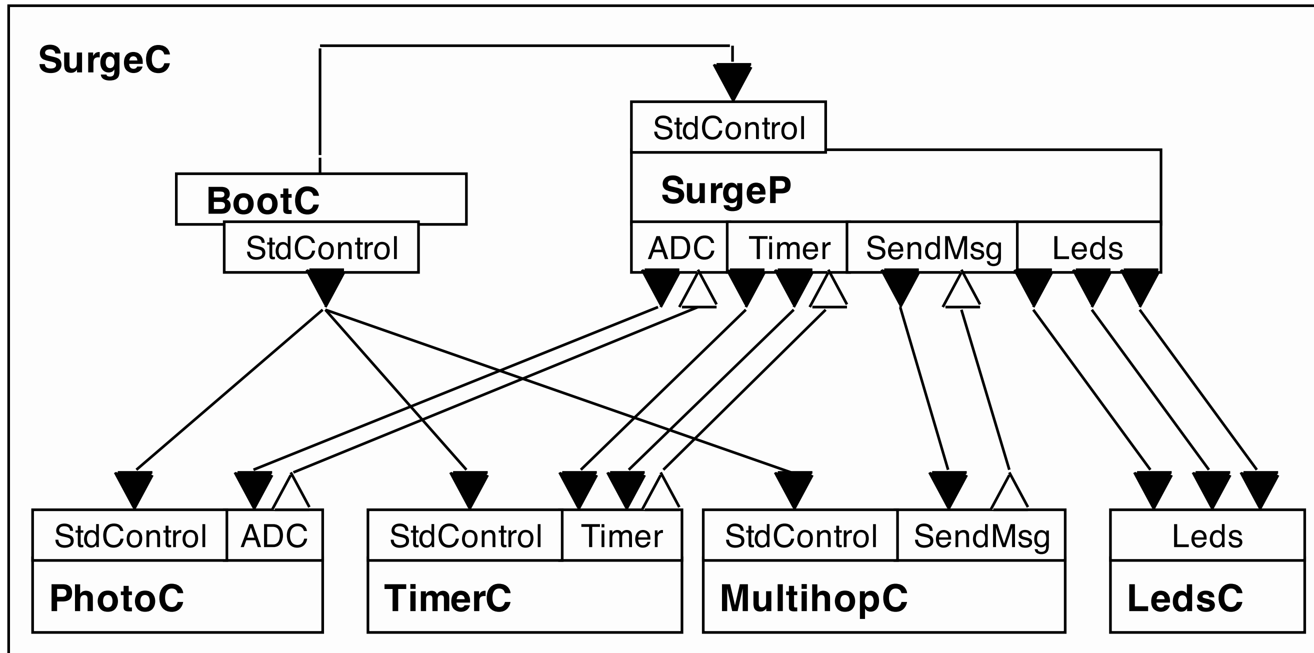
Configurations

- Select & wire together modules




```
configuration TimerC {
  provides {
    interface StdControl;
    interface Timer;
  }
}
implementation {
  components TimerP, HWClock;

  StdControl = TimerP.StdControl;
  Timer = TimerP.Timer;

  TimerP.Clock -> HWClock.Clock;
}
```

Example: Surge



Concurrency

- Race condition: concurrent interrupts/tasks update shared variables.
- Asynchronous code (AC): reachable from at least one interrupt handler.
- Synchronous code (SC): reachable from tasks only.
- Any update of a shared variable from AC is a potential race condition.

A Race Condition

```
module SurgeP { ... }
implementation {
    bool busy;
    norace uint16_t sensorReading;
    async event result_t Timer.fired() {
        if (!busy) {
                        busy = TRUE;
                        call ADC.getData();
        }
        return SUCCESS;
    }
    task void sendData() { // send sensorReading
        adcPacket.data = sensorReading;
        call Send.send(&adcPacket, sizeof
        adcPacket.data);
        return SUCCESS;
    }
    async event result_t ADC.dataReady(uint16_t data) {
        sensorReading = data;
        post sendData();
        return SUCCESS;
    }
```

Atomic Sections

```
atomic {
    <Statement list>
}
```

- Disable interrupt when atomic code is being executed
- But cannot disable interrupt for long!
  - No loop
  - No command/event
  - Function calls OK, but callee must meet restrictions too

Prevent Race

```
module SurgeP { ... }
implementation {
    bool busy;
    norace uint16_t sensorReading;

    async event result_t Timer.fired()
    {
        bool localBusy;
        atomic {
              localBusy = b
              busy = TRUE;
        }
        if (!localBusy)
              call ADC.getData();
        return SUCCESS;
    }
```

disable interrupt

enable interrupt

test-and-set

nesC Compiler

- **Race-free invariant**: Any update to a shared variable is
  - from synchronous context only, or
  - occurs within an atomic section.
- Compiler returns error if the invariant is violated.
- Fix
  - Make access to shared variables atomic.
  - Move access to shared variables to tasks.

Results

- Tested on full TinyOS code, plus applications
  - 186 modules (121 modules, 65 configurations)
  - 20-69 modules/app, 35 average
  - 17 tasks, 75 events on average (per application)
    - Lots of concurrency!
- Found 156 races: 103 real!
  - About 6 per 1000 lines of code
- Fixing races:
  - Add atomic sections
  - Post tasks (move code to task context)

Optimization: Inlining

App	Code size		Code reduction	Data size	CPU reduction
	inlined	*noninlined*			
Surge	14794	16984	12%	1188	15%
Maté	25040	27458	9%	1710	34%
TinyDB	64910	71724	10%	2894	30%

- Inlining improves performance and reduces code size.
- Why?

Overhead for Function Calls

- Caller: call a function
  - Push return address to stack
  - Push parameters to stack
  - Jump to function
- Callee: receive a call
  - Pop parameters from stack
- Callee: return
  - Pop return address from stack
  - Push return value to stack
  - Jump back to caller
- Caller: return
  - Pop return value

Principles Revisited

- Support TinyOS components
  - Interface, modules, configuration
- Whole-program analysis and optimization
  - Improve robustness: detect race conditions
  - Optimization: function inlining
  - More: memory footprint.
- Static language
  - No malloc, no function pointers

Issues

- Acceptance of a "new" programming language?

- No dynamic memory allocation
  - Bound memory footprint
  - Allow offline footprint analysis
  - But how to size buffer when data size varies dynamically?

- Restriction: no "long-running" code in
  - Command/event handlers
  - Atomic sections

Reading

- D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler, The nesC Language: A Holistic Approach to Networked Embedded Systems. [Required]
- J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister, System Architecture Directions for Network Sensors.
- P. Levis and D. Gay, TinyOS Programming, Cambridge University Press, 2009.
  - Purchase the book online
  - Download the first half of the published version for free.
- http://www.tinyos.net/