Operating Systems

Sensor Networks

Prof. Dr. Kay Römer

OS Functionality

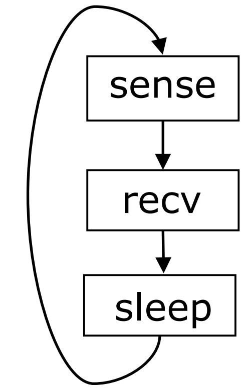
- Abstraction
  - Simplified access to hardware
- Management of resources
  - RAM, disks
  - Processor
  - Communication / network
  - Input / output
- How do sensor nodes differ from other computers?
  - Severely limited resources
    - Energy, memory, computing power
  - „Simplistic“ processor
    - No memory protection
    - No virtual memory
  - Very specific application properties
    - „Sense, Process, Send“

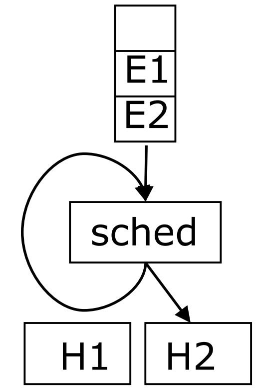
Concurrency

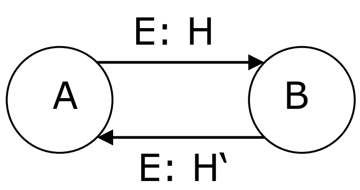
- Quasi-parallel execution of multiple tasks
  - Read sensors, receive messages, ...
- Traditional OS abstractions
  - Process: state (context), stack, address space
  - Thread: state (context), stack
- Problems
  - Memory: stack, address space
  - Overhead process/thread context change
- But: often only single application on sensor node!
  - Real need for processes?

Concurrency in SN

- „While-Loop“
  - Sequentially execute all tasks in a loop
- Events
  - Asynchronous events (message reception, timeout) trigger execution of event handler function
  - Run-to-completion
- Prioritized events
  - Priority 1 events interrupt execution of priority 2 event handlers
  - Cf. hardware interrupts
- State machines
  - Reaction depends on state and event
- Lightweight threads
  - Typically not preemptive
  - Typically only one shared stack
- Hybrid variants





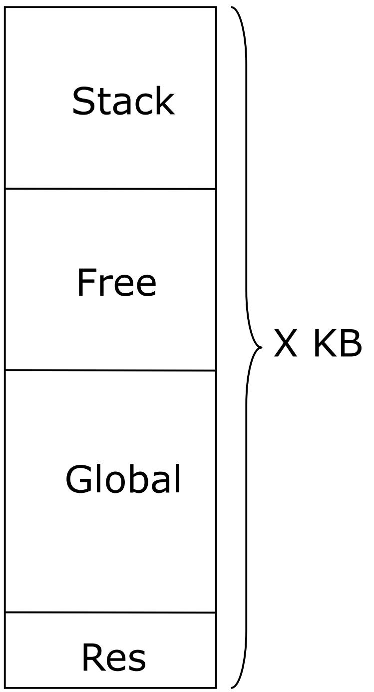


Memory Management

- Efficient allocation of memory to tasks and OS and its protection
- Traditional OS abstractions
  - Virtual address space per process
  - Dynamic mapping of physical memory to virtual address space
  - Protection mechanisms
    - Among processes and OS
- SN challenges
  - Lack of hardware support for virtual memory management and protection
- But: typically only one process

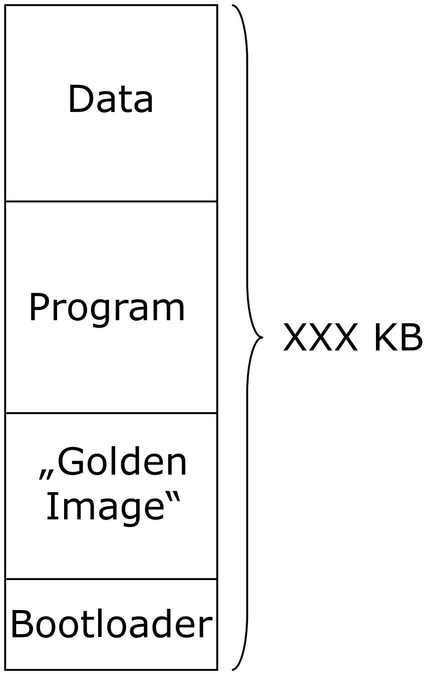
Memory Management in SN

- Simplified memory model
  - Reserved area
    - System parameters
  - Global variables
  - Stack
  - Typically no dynamic memory allocation
    - Local variables (stack)
    - Global variables
- Typically no separation of OS and application



Secondary Memory

- Traditional OS
  - Disks, flash, ...
  - File systems
- Sensor networks
  - Typically multiple flash memories
    - Program (up to 100 kB, part of microcontroller)
    - Data (up to multiple MB, separate chip)
  - Typically random access, more recently simple file systems
  - Program flash
    - Firmware
    - Application code
    - Application data

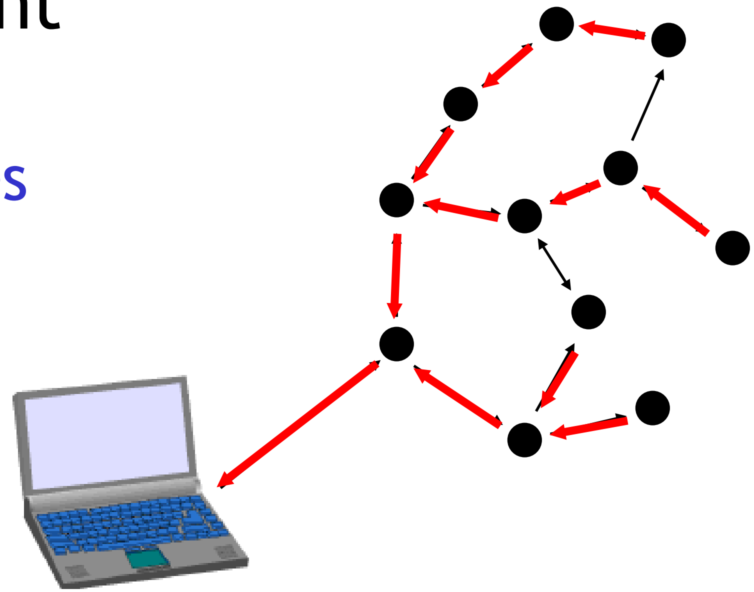


Communication

- Exchange of messages among sensor nodes
- Traditional OS
  - Multiple network interfaces, multiple protocol stacks
  - Typically TCP/IP with socket interface
- SN challenges
  - Memory overhead (buffer, code)
  - Different requirements
    - TCP: end-to-end delivery with guarantees
    - WSN: hop-by-hop, data processing in the network, more focus on efficiency than on reliability
    - But: increasing importance of IPv6 (6LoWPAN)

Communication in SN

- Medium
  - Radio
- Media access (MAC)
  - Often CSMA-based
- Neighborhood management
  - Link quality estimation
  - Selection of neighbor nodes
- Multi-hop routing
  - Often tree based
    - Sink to all nodes
    - Nodes to sink

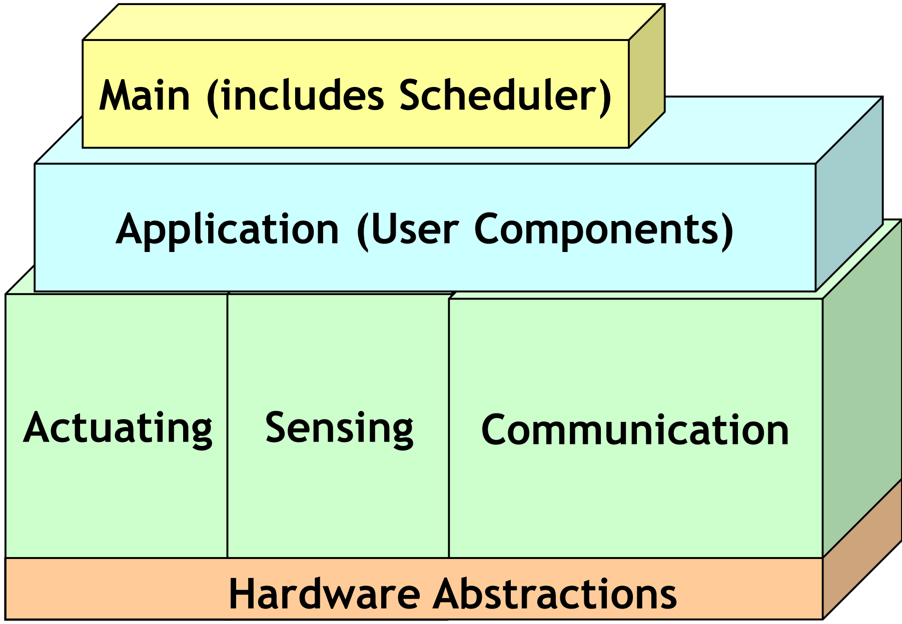


Other OS Functions

- Drivers
  - Sensors, real-time clock, LED, I/O (serial, busses)
- Energy management
  - Enable/disable subsystems, energy saving modes
  - Duty-cycling
- Programming „over-the-air“
  - Distribute program images via radio and reprogram nodes
- Component-based programming models
  - Modularity and reuse (replacement for processes)
  - Libraries of code modules

Case Study: TinyOS

- Popular operating system of Berkeley Motes
- Key abstractions
  - Component system
  - Prioritized events + scheduler
  - Active messages („Remote Events“)
- nesC programming language
  - C extension / subset
  - Language support for component system
  - Preprocessor
- Component library
  - Main
  - Hardware abstractions
  - Sensor stack
  - Actuator stack
  - Communications stack



Components: Overview

- Commands
  - Function (cf. method in OO)
- Event
  - HW interrupt or signaled by program
- Event handler
  - Short function to be executed upon occurrence on an event
- Task
  - Longer, asynchronously executed function

Commands

- Cf. method invocation

```
// Component A

{
  ...
  status = call CmdName(args)
  ...
}
```

```
// Component B

command CmdName (args) {
  ...
  return status;
}
```

Events + Event Handler

- Event triggers
  - Hardware interrupts
  - Generated by application using „signal“
- Handler function defined for each event
  - Run-to-completion
  - Preempt tasks
  - Short runtime (interrupts!)

```
// Component A

{
  ...
  status = signal EvtName (args)
  ...
}
```

```
// Component B

event EvtName (args) {
  ...
  return status;
}
```

Tasks

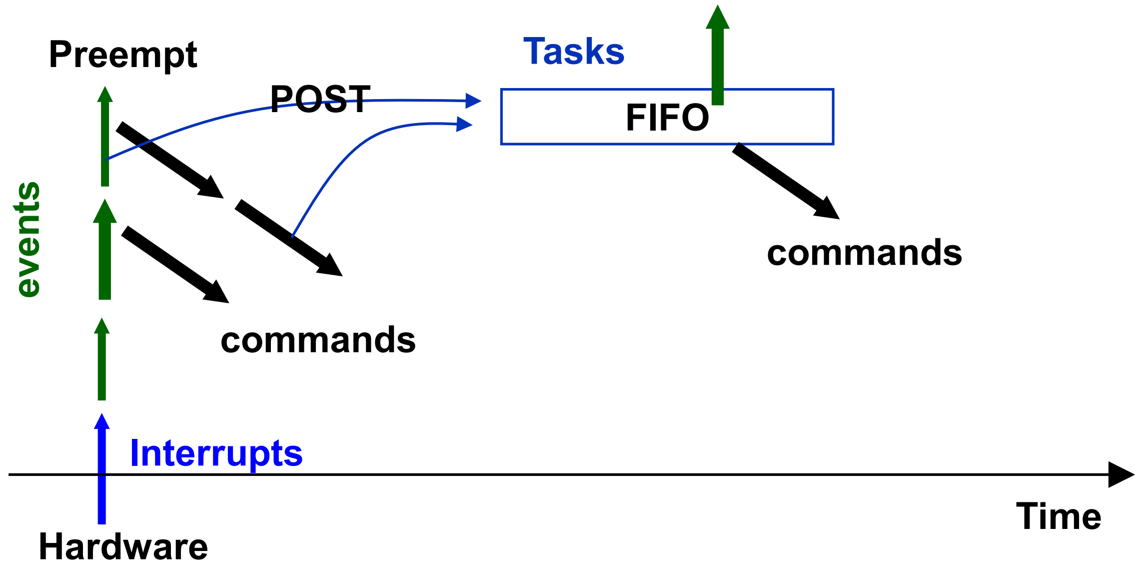
- Longer, asynchronously executed functions
  - Preempted by events
  - Not preempted by other tasks
  - Execute commands, signal events
  - Cf. prioritized events
- Scheduling
  - FIFO using a task queue
  - Sleep upon empty queue
- Typical code pattern: split-phase operation
  - Event handler posts task to do work
  - When task finished, signal event

```
{
  ...
  post TaskName();
  ...
}
```

→

```
task TskName (args) {
  ...
}
```

Tasks, Events, Commands



Components

- Interface
  - Defines a set of commands and events
- Module
  - Implementation of one or more interface(s)
- Configuration
  - Composition of multiple modules

Interface

```
interface StdControl {
  command result_t init();
  command result_t start();
  command result_t stop();
}
```

```
interface Timer {
  command result_t start (...);
  event result_t fired();
}
```

Modules

- Module
  - Offer interface implementations to other modules („provides“)
    - Implement the provided interfaces
  - Uses interfaces provided by other modules („uses“)
  - Naming convention: *M

```
Module SurgeM {
  provides interface StdControl;
  uses interface ADC;
  uses interface Timer;
  uses interface Send;
} implementation { ... }
```

Configuration

- Composition of multiple modules
  - New module (naming convention: *C)
  - Complete application
- Implementation
  - Connect «provided» interface of one module to «used» interface of other module

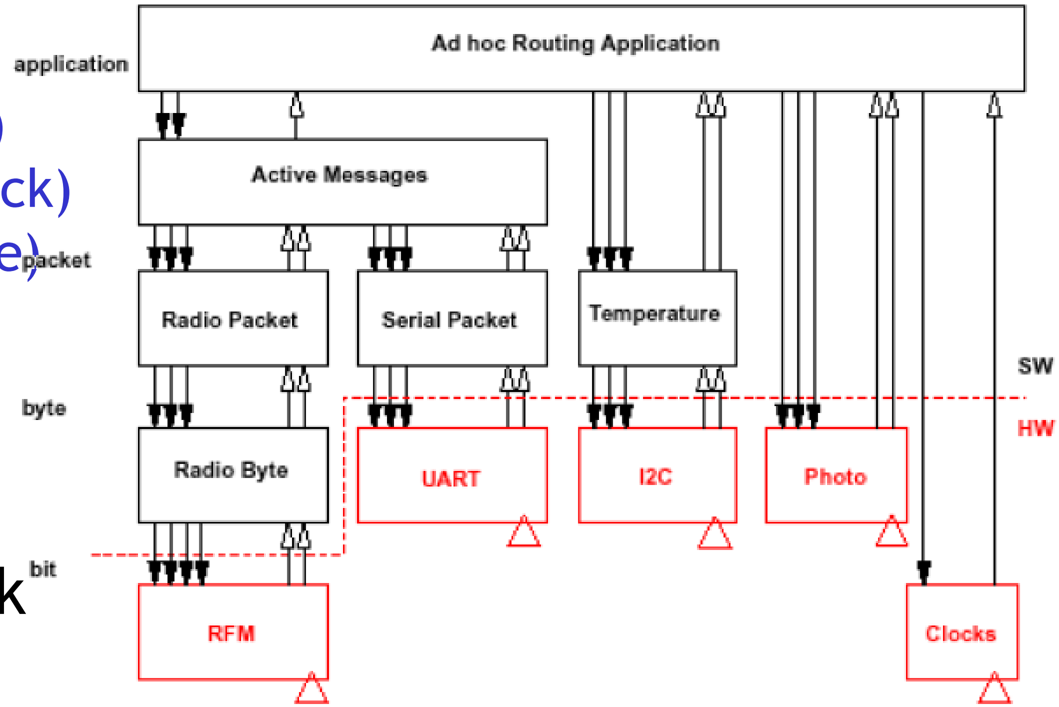
```
configuration Surge {
} implementation {
  components Main, SurgeM, TimerC, Photo, MessageQueue;
  Main.StdControl -> SurgeM.StdControl;
  Main.StdControl -> TimerC.StdControl;
  SurgeM.Timer -> TimerC.Timer;
  ...
}
```

Active Messages (AM)

- Communication among sensor nodes
  - Remote invocation of event handler
- AM content
  - Identification of event (ID)
  - Payload / event parameters
- Upon AM reception
  - Map event ID -> Event (node specific)
  - signal Event (payload)

Component Library

- HW abstractions
  - LED (light em. diode)
  - CLOCK (real-time clock)
  - UART (serial interface)
  - ADC (A/D converter)
  - RFM (bit-level radio)
- Sensor stack
  - Sensors: temp., …
- Communication stack
  - Bit level (radio abstraction)
  - Byte level (bits -> byte)
  - Packet level (bytes -> packet)
  - Active messages (remote events)



Example: LED Hello World

```
/** HelloWorldM.nc by Gary Wong */

#define MORSE_WPM 12 /* speed, in words per minute */
#define MORSE_UNIT ( 1200 / MORSE_WPM ) /* ms per unit */

module HelloWorldM {
   provides {
      interface StdControl;
   }
   uses {
      interface Timer;
      interface Leds;
   }
} implementation {
   command result_t StdControl.init() {
      call Leds.init();
      call Leds.redOff();
      call Leds.yellowOff();
      call Leds.greenOff();
      return SUCCESS;
   }

   command result_t StdControl.start() {
      return call Timer.start( TIMER_ONE_SHOT, 1000 );
   }

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

   ...
```

```
event result_t Timer.fired() {
      static char *morse = ".... . .-.. .-.. --- --..--  “;
      static char *current = 0;
      if( !current )  current = morse;

      switch( *current ) {
      case ' ': /* pause: off for two units */
         call Timer.start( TIMER_ONE_SHOT, 2 * MORSE_UNIT );
         current++;
         break;
      case '.': /* dot: on for one unit, off for one unit */
         if( !call Leds.get() ) {
             call Leds.redOn();
             call Timer.start( TIMER_ONE_SHOT, MORSE_UNIT );
         } else {
             call Leds.redOff();
             call Timer.start( TIMER_ONE_SHOT, MORSE_UNIT );
             current++;
         }
         break;
      case '-': /* dash: on for three units, off for one unit */
         if( !call Leds.get() ) {
              call Leds.redOn();
              call Timer.start( TIMER_ONE_SHOT, 3 * MORSE_UNIT );
         } else {
              call Leds.redOff();
              call Timer.start( TIMER_ONE_SHOT, MORSE_UNIT );
              current++;
         }
         break;
      }
      /* wrap around string if end reached */
      if( !*current ) current = morse;

      return SUCCESS;
   }
}
```

Example: LED Hello World

```
/*
 * HelloWorld.nc
 *
 * by Gary Wong
 *
 */

configuration HelloWorld {
}
implementation {
   components Main, HelloWorldM, TimerC, LedsC;

   Main.StdControl -> HelloWorldM.StdControl;
   Main.StdControl -> TimerC.StdControl;

   HelloWorldM.Timer -> TimerC.Timer[ unique("Timer") ];
   HelloWorldM.Leds -> LedsC;
}
```

- Memory
  - Code: ~ 2kB
  - RAM: ~ 70 Bytes

- **Creates Timer instance**
- **„unique" computes unique index**

Other SN OS

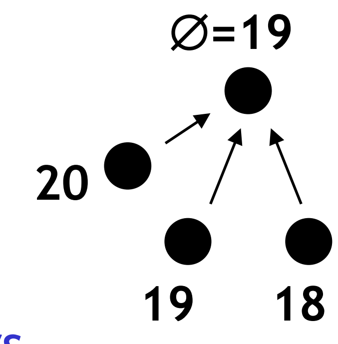
- Contiki
  - Plain C
  - Dynamic loading of modules
  - Events
  - Protothreads: non-preemptive, stack-less threads
  - uIP, sicsLowPan: IPv4 and IPv6 implementations
- MANTIS
  - Separation kernel / application
  - Preemptive multi-threading

Events: Friend of Foe?

- Events vs. threads (religious war)
  - Events: efficient but complex to use
  - Threads: easy to use but inefficient
- Event-based programming common in SN
  - Due to limited resources
- Event problems
  - Splitting function into many event handlers
  - Implication 1: Global variables
  - Implication 2: Explicit states

Events: Global Variables

- Example: Collect sensor values and compute average
  - Collect sensor values from neighbors
  - After timeout compute average of all values
- Split into two event handlers
  - Global variables for data exchange among handlers
  - Control flow not visible, error-prone
  - Waste of memory

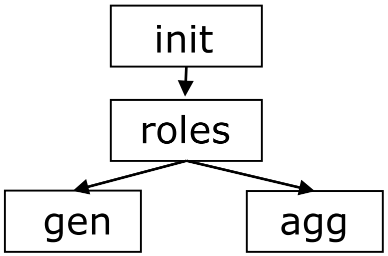



```
// Thread

void aggregate () {
  int sum = 0;
  int num = 0;

  while (!timeout()) {
    MSG msg = recv();
    sum += msg.val;
    num += 1;
  }
  send (sum/num);
}
```

```
// Events

int sum = 0;
int num = 0;

void recv_handler (MSG msg) {
  sum += msg.val;
  num += 1;
}

void timeout_handler () {
  send (sum/num);
}
```

Events: Explicit States

- Example: program phases
  - Initialization, role assignment, data generation OR data aggregation
- Explicit states
  - State machines in each event handler
  - Error-prone, modularity?

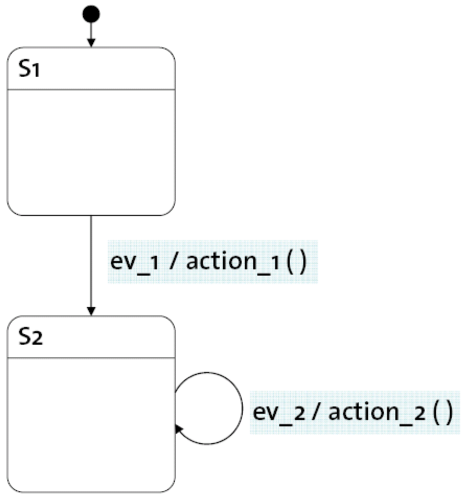



```
// Thread

void main () {

  { // init ... }

  { // roles ... }

  if (role == gen) {
    // gen ...
  } else {
    // agg ...
  }
}
```

```
// Events

int state = INIT;

void handler (event e) {
  switch (state) {
  case INIT:
    ...
    state = ROLE;
    ...
  case ROLE: ...
  case GEN: ...
  case AGGR: ...
}
```

OSM: Extended FSM

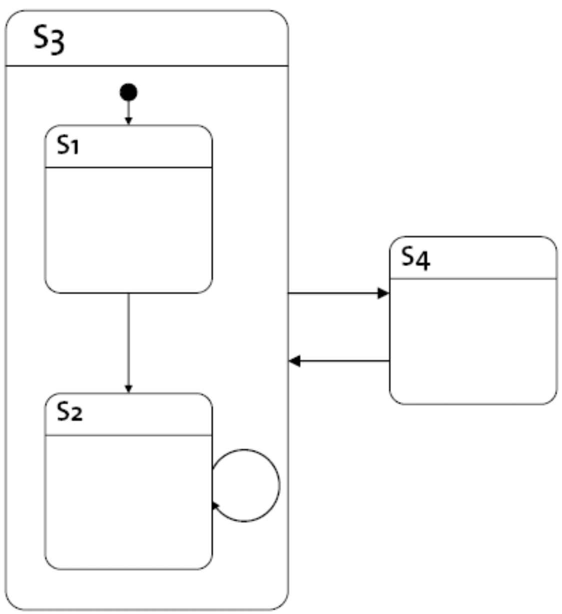
- Language for FSM
  - Based on StateCharts
  - Modularity by composing FSM
  - Compiler translates OSM to C, almost as efficient as event-based code
- Resolves problem with global variables
  - States can have local variables

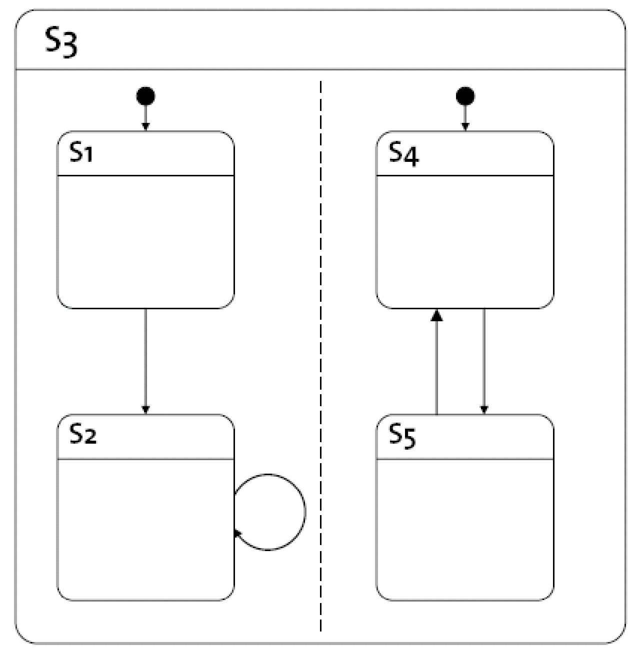



```
state S1 {
  ev_1 / action_1() -> S2;
}

state S2 {
  ev_2 / action_2() -> S2;
}
```

OSM: Composition

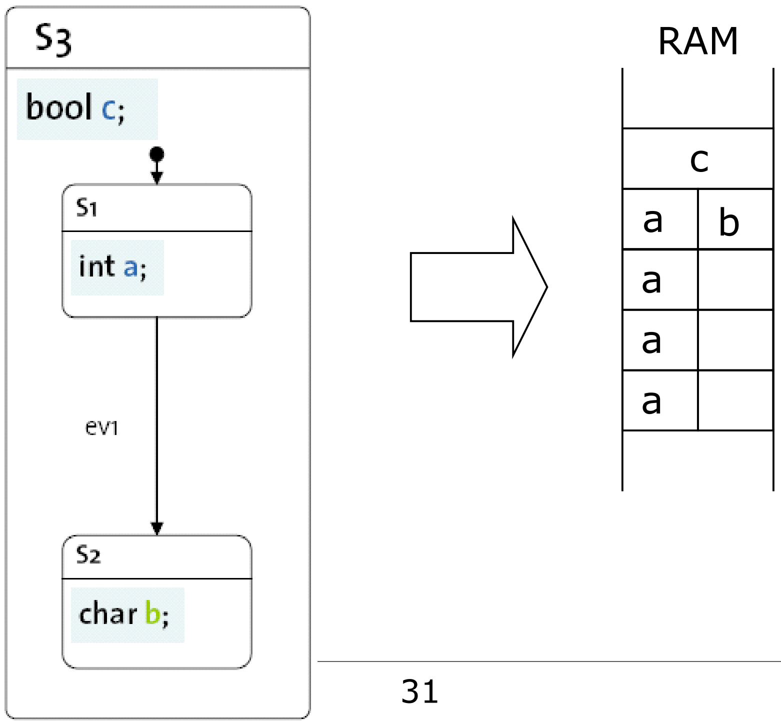
- Composition of FSM
  - Hierarchical: State contains FSM
  - Parallel: Concurrent FSM





OSM: Variables

- States can contain variables
  - Access only while containing state is active
  - Compiler overlays variables that cannot be active simultaneously
  - Without dynamic memory allocation



Protothreads

- Light-weight threads
  - Wait for event in event handler
  - Single shared stack
- Code structure similar to traditional threads
  - Problem with global variables unsolved

```
// Thread

void aggregate () {
  int sum = 0;
  int num = 0;

  while (!timeout()) {
    MSG msg = recv();
    sum += msg.val;
    num += 1;
  }
  send (sum/num);
}
```

```
// Protothread

void handler (event e) {
  static int sum = 0;
  static int num = 0;

  PT_BEGIN
  while (true) {
    PT_WAIT (e == recv || e == timeout);
    if (e == recv) {
      sum += msg.val;
      num += 1;
    }
    if (e == timeout) {
      send (sum / num);
      return;
    }
  }
  PT_END
}
```

global variable

Protothreads: Approach

- PT_WAIT does NOT block
- If condition false, leave event handler with return
  - Save position of PT_WAIT
- Next event invokes event handler again
  - Jump to saved position
- Content of local variables (on stack) lost!

Protothreads: Implementation

- C preprocessor macros

```
#define PT_BEGIN   static int state = 0; switch(state) { case 0:
#define PT_END     }
#define PT_WAIT(c) state = __LINE__; case __LINE__: if (!(c)) return;
```

```
void handler (event e) {
  static int sum = 0;
  static int num = 0;

  PT_BEGIN


  while (true) {
    PT_WAIT (e == recv || e == tout);


    if (e == recv) {
      sum += msg.val;
      num += 1;
    }
    if (e == timeout) {
      send (sum / num);
      return;
    }
  }
  PT_END
}
```

line 9

```
void handler (event e) {
  static int sum = 0;
  static int num = 0;

  static int state = 0;
  switch (state) {
  case 0:
    while (true) {
      state = 9;
  case 9:
      if (!(e == recv || e == tout)) return;
      if (e == recv) {
        sum += msg.val;
        num += 1;
      }
      if (e == timeout) {
        send (sum / num);
        return;
      }
    }
  }
}
```

References

- Slides contain material of the following authors
  - Joe Polastre – Berkeley
  - Adam Dunkels – SICS
  - Rick Han – Boulder
  - Oliver Kasten, Matthias Ringwald – ETH Zurich
  - Chenyang Lu – Virginia