SES
Chapter 6: Operating Systems for Embedded Systems

Prof. Dr.-Ing. Bernd-Christian Renner

Institute smartPORT
Hamburg University of Technology

TUHH

Contents

Implementing Embedded Systems

- Embedded systems are often reactive systems
  - they respond to external events
  - often require real-time response
- Simplest structure for reactive programs: Super-loop
  - Functional related operations are grouped into tasks which are strung together in a large endlessly executing loop
  - Each task
    - is coded as a separate block of code with a single start and a single end point (e.g. as a single function)
    - first checks for a relevant state change and if necessary executes code
- Completely deterministic model, i.e. response time can be calculated (if execution times are fixed)

1

Schedulers

Overview

- Higher level:
  - Scheduler allows tasks to be called (periodically or sporadically)
- Lower level:
  - Scheduler can be viewed as a single timer interrupt service routine that is shared between many different tasks
    - i.e. only one timer needs to be initialized, any changes to the timing generally requires only one function to be altered
- Scheduler is independent of nature/number of tasks
  - Advantage: Scheduler has to be written only once
- Scheduler: Beginning of operating system

Round Robin Scheduler

- Round Robin Scheduler (RRS)
  - Scheduler maintains a list of all active tasks (task list)
  - Tasks operate on shared data
  - RRS takes care of switching from one task to the next
  - Programmer has to initially tell scheduler which tasks are to run, but after that scheduler takes control
  - All tasks run to completion (not preemptive)
- Design of a simple RRS
  - Implement scheduler
  - Declare prototypes for tasks
  - Implement tasks

Tasks

- Each task must have the same signature
- Generic case
  - Return type `void`
  - One parameter of type `void *` (can also be used to return values)
- Tasks are stored in array

```
void (*queue[TASKCOUNT])(void *);
```

- Example: Three tasks (parameter is address of shared data)

```
void get(void * aNumber);          // input task
void increment(void * aNumber);    // computation task
void display(void * aNumber);      // output task
```

Implementation of Tasks

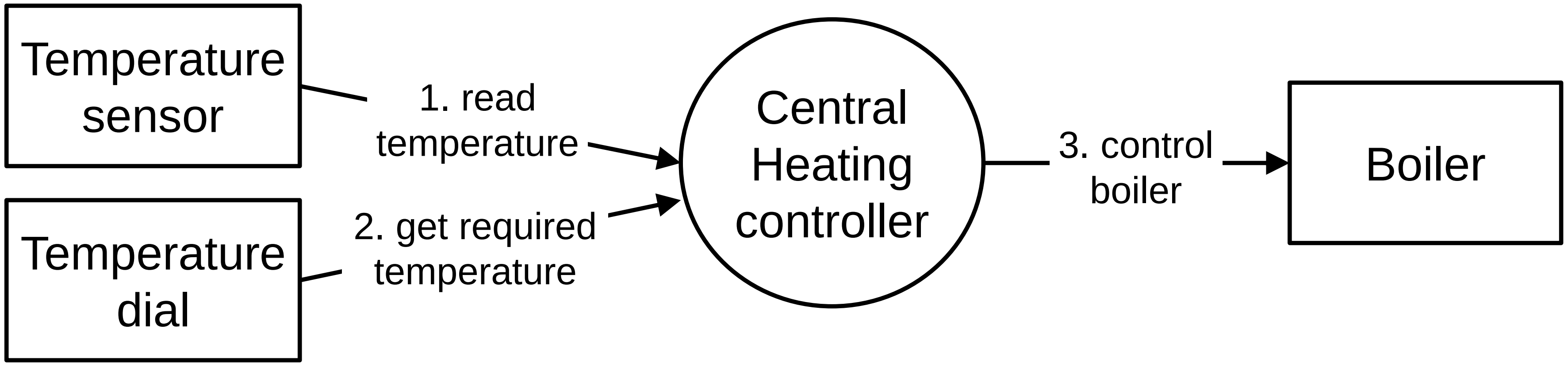
```
void get(void * aNumber) {                    //input task
        int * intPtr = (int *)aNumber;
        printf(“Enter a number: 0..9”);
        *intPtr = getchar();
        getchar();
        *intPtr -= ‘0’;
        return;
}

void increment(void * aNumber) {              // computation task
        int * intPtr = (int *)aNumber;
        (*intPtr)++;
        return;
}

void display(void * aNumber) {                // output task
        printf(“The result is: %d\n”, *(int *)aNumber);
        return;
}
```

RRS: Code for Scheduler

```
void main() {
        int i = 0;                              // queue index
        int data;                               // shared data
        int * intPtr = &data;                   // pointer to data

        void (*queue[TASKCOUNT])(void *);       // task queue

        queue[0] = get;
        queue[1] = increment;
        queue[2] = display;

        while (1) {                             // scheduler
                queue[i]((void*) intPtr);       // dispatch task
                i = (i+1) % TASKCOUNT;
        }
        return;
}
```

Example: Central-heating controller

- 3 Tasks
- Fixed order
- Round Robin Scheduler is simple solution



Limitations

- Tasks are run in a fixed order
  - Tasks cannot be skipped
  - No priorities
- No control over timing of tasks
  - A task has to wait until all preceding tasks have finished
  - Execution period of a task cannot be predicted
    - task execution times may vary
- Operates at ‘full power’ at all times
- No ad-hoc execution of a task
- Extensions:
  - Dynamically add, remove, stop, and resume tasks
  - Excecute tasks upon events
  - Execute tasks at specific times

Alternative: Event Driven Loop

- Tasks are executed upon events
- All tasks are implemented as ISRs
- Main program:
  - infinite loop interrupted when event occurs
  - flow of control jumps to associated ISR, where designated task is executed
  - afterwards flow resumes infinite loop
- More difficult to analyze due to asynchronous events
- Often used with priorities to make it more deterministic

```
global variable declarations

isr set up
function prototypes
void main (void)
{
    local variable declarations
    while(1);          // task loop
}
ISRs
function definitions
```

Periodic Scheduler

- Tasks are executed periodically
- Example:
  - Tasks are finite state machines (FSM), i.e., each task is implemented by a function with signature
    `uint8_t (*func)(uint8_t)`
  - Each task can have its own execution period (multi-rate)
- Data structure for tasks

```
typedef struct {
   uint8_t state;
   uint32_t period;
   uint32_t elapsedTime; // time since last execution
   uint8_t (*func)(uint8_t)
} task_t;
```

Periodic Scheduler

- Initialization

```
task_t task_1, task_2, …;
task_t * tasks[] = {&task1, &task2, …}
const uint8_t numTasks = sizeof(tasks)/sizeof(tasks[0]);

//Init each task
task_1.state = -1;  // initial state of FSM
task_1.period = …;
task_1.elapsedTime = task_1.period;
                 // all tasks are immediately executed
task_1.func = func1;
...

//ISR for timer
volatile uint8_t timerFlag = 0;

void timerISR() {
    timerFlag = 1;
}
```

Periodic Scheduler

- Code for Scheduler

```
timerSet(basePeriod);
timerOn();

while (1) {
   for (i=0; i < numTasks; i++) {
      if (tasks[i]->elapsedTime == tasks[i]->period) {
         task[i]->state =
            tasks[i]->func(tasks[i]->state);
         tasks[i]->elapsedTime = 0;
      }
      tasks[i]->elapsedTime += basePeriod;
   }
   while (timerFlag == 0);
   timerFlag = 0;
}
```

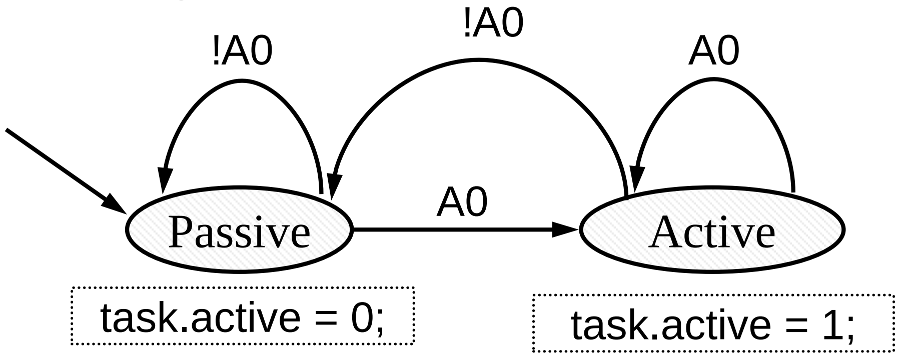
All tasks together must complete within basePeriod

Aperiodic tasks

- Not all tasks are periodic tasks
  - Example: A system has to monitor an input source for occurrence of a special value
- Possible solution: Periodic polling
  - Con: High microcontroller usage
- Alternative:
  - Some microcontrollers come with special hardware to detect changes on an input and call an ISR
  - Example: UART provides such events
- New concept: Asynchronous finite state machine
  - A task is triggered by an event and not executed periodically

Scheduler for periodic and aperiodic tasks

- Extend data structure for task by
  `volatile uint8_t active;`
- Model each monitoring task as



- For each monitoring task provide separate ISR

```
void eventA0_ISR() {
   tasks[i]->active = 1;
   tasks[i]->elapsedTime = tasks[i]->period;
   eventFlag = 1;
}
```

- Task `tasks[i]` handles event `A0`

Main Loop

```
volatile unit8_t eventFlag = 0;
while (1) {
   for (i = 0; i < numTasks; i++) {
      if (tasks[i]->active) {
         if (tasks[i]->elapsedTime == tasks[i]->period) {
            task[i]->state = tasks[i]->func(tasks[i]->state);
            tasks[i]->elapsedTime = 0;
            if (tasks[i] is non-periodic task) {
               tasks[i]->active = 0;
            }
         }
         tasks[i]->elapsedTime += basePeriod;
      }
   }
   while (timerFlag == 0 && eventFlag == 0) { /* busy wait */ };
   timerFlag = 0;
   eventFlag = 0;
}
```

Advanced Schedulers

- Schedulers considered so far let each task run to completion before next task is started
- Alternative:
  - Execution of a task can be halted and another task begins execution
  - At a later point execution of first task is resumed
- Realizations:
  1. Cooperative scheduler
     - **Tasks** decide themselves when to halt and yield to another task
  2. Preemptive scheduler
     - **Scheduler** decides when to halt current task and resume another task
- Changing of running task is called *context switch*

Advanced Schedulers

- Dispatcher:
  - Component that is responsible for remembering yielding task's whereabouts (called task’s execution context) and starting up next task from task set
  - Execution context is important for resuming the task
    - contents of all registers (including program counter) and stack
  - Tasks of dispatcher
    - Save execution context of current task in RAM
    - Load execution context of next task from RAM into registers and restore stack
- Scheduler:
  - Decides which task to run next
  - May use priorities
  - Requires data structure to maintain tasks

Cooperative Schedulers

- Operation of a cooperative scheduler:
  - Task executes a special instruction that lets it yield control back to scheduler (implemented as a software interrupt)
  - Scheduler selects task to run next
  - Scheduler advices dispatcher to do a context switch
    - Next task starts off at exactly where it left off
- When does a task yield?
  - it has done all it can for now
  - taken up enough processor time
- Problems with cooperative schedulers
  - What happens if task *forgets* to yield?
  - What happens if a task crashes?

Preemptive Schedulers

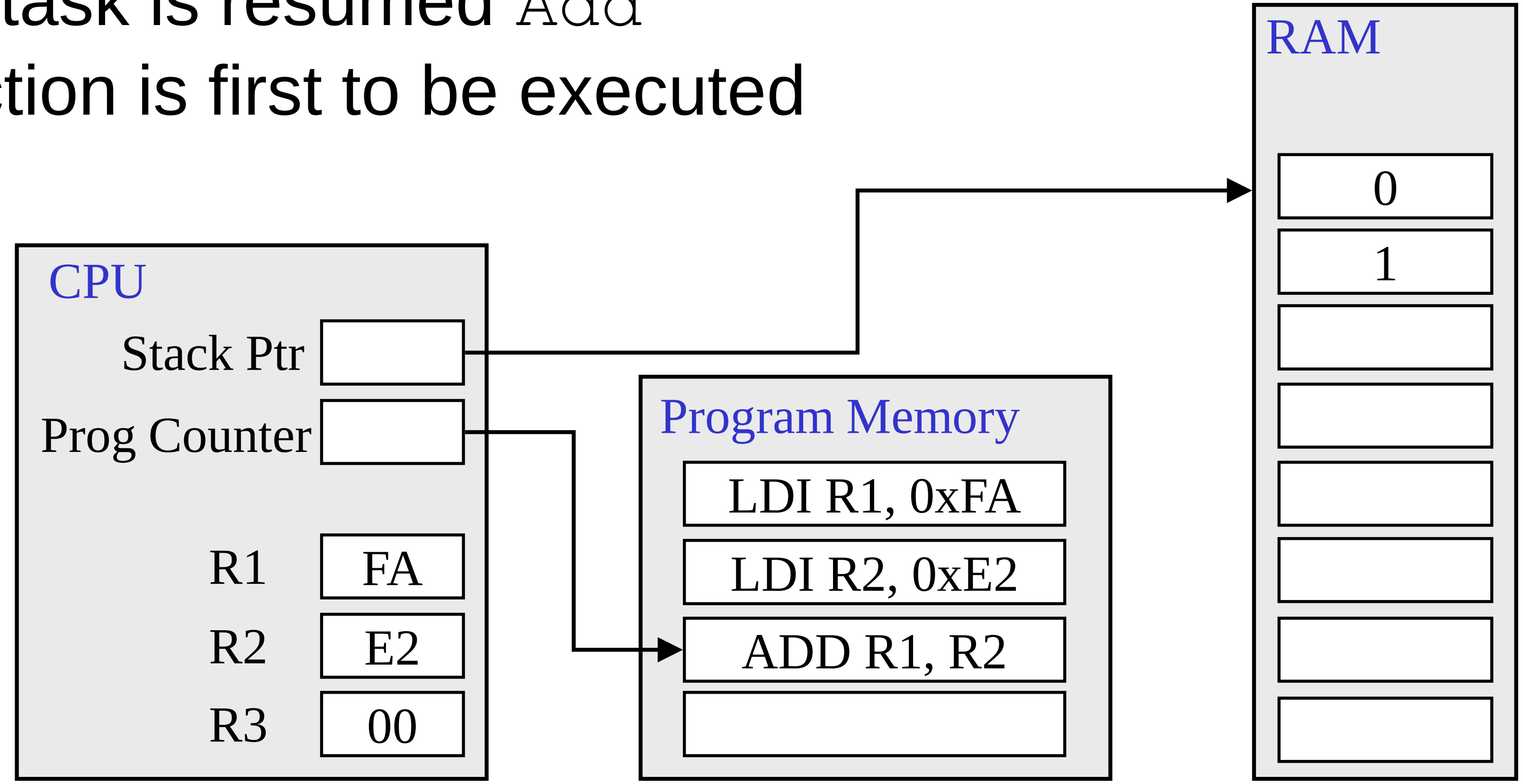
- In preemptive scheduling tasks can be **forced** to interrupt, i.e. tasks are not yielding voluntarily
- Context switches are triggered by events
  - externally generated by hardware inputs
    - e.g. reading from a UART
  - internally generated by hardware timers
- Interruption of tasks is transparent for users
- Preemptive schedulers also require a dispatcher, same duties as with cooperative scheduler

Execution Context

- Task does **not** know when it is going to be suspended or resumed
- While task is suspended other tasks will execute and may modify register values
- It is essential that upon resumption a task has a context identical to that immediately prior to its suspension
- Context switch is based on periodic timer
  - Timer ISR checks if a context switch must be performed
  - Context switch is implemented by dispatcher
- Dispatcher realized as ISR

Example

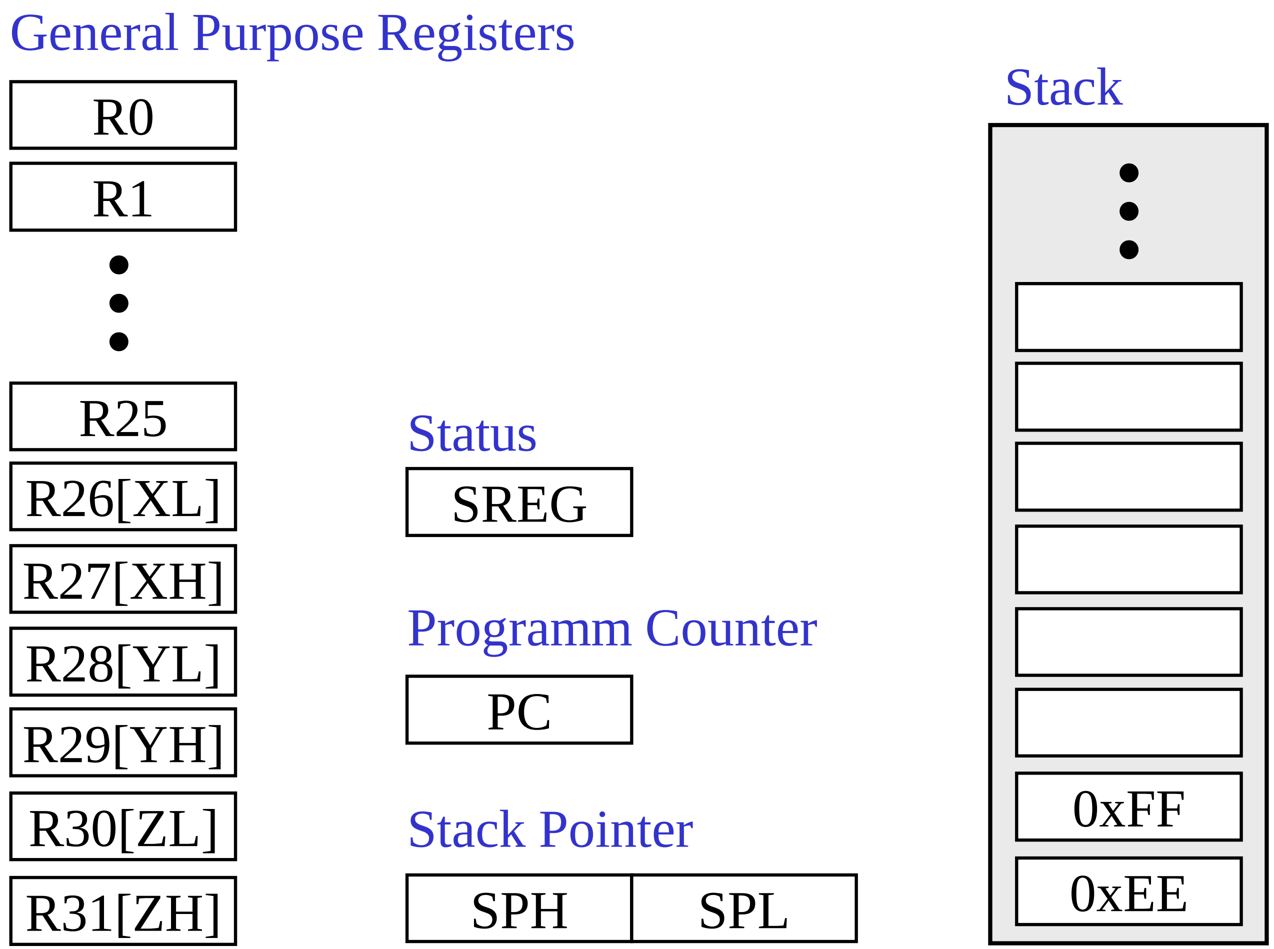
- Task being suspended is immediately before executing an instruction adding values contained in registers R1 and R2. When task is resumed `Add` instruction is first to be executed



AVR Context

- AVR Context:
  - 32 general purpose registers
  - Status register
    - Its value affects instruction execution, and must be preserved across context switches
  - Program counter
    - Upon resumption, a task must continue execution from the instruction that was about to be executed immediately prior to its suspension
  - Two stack pointer registers
  - Stack

AVR Execution Context



ISR and Execution Context

- Code of ISR must save and restore context
- Compiler generated ISR ensures that **every register that gets modified by ISR** is restored to its original value when ISR exits
  - Upon ISR completion:
    Always return to code when interrupt occurred
  - Context switch is different:
    Compiler cannot make assumptions as to when context switch will happen, and therefore cannot optimize which registers require saving and which don't (any code could be executed next)
- Consequence:
  Entire context must be saved and context of next task must be loaded

Saving the Context

- Generated ISR code forces a *return from interrupt* instruction (RETI) to be used in place of the 'return' instruction (RET) that would otherwise be used
  - AVR microcontroller disables interrupts upon entering an ISR and RETI instruction is required to re-enable them on exiting
- Context switch requires Dispatcher to explicitly save all registers on entering ISR, but doing so would result in some registers being saved twice
  - once by compiler generated code and again by dispatacher code
- Solution: `naked` attribute
  - prevents compiler from generating any function entry or exit code so this must be added explicitly
  - gives application code complete control over when and how context is saved/restored

Saving the Context

- Context is pushed on stack
  - program counter is pushed by ISR automatically
  - r0 to r31, SREG, and stack pointer must be pushed manually
- Each task requires its own stack, when a task swap is performed, pointers to corresponding stack are restored
- Allocate for each task to be executed (how many?) memory (how much?) to host task's stack and to store stack pointer
- This structure is called **task control block** (TCB)
- When task is
  - preempted: push context on stack, write stack pointer variable
  - resumed: restore stack pointer, pop and restore context from stack
- Memory for each stack is fixed, so beware of stack overflow!

Context Switching ISR

```
void IST_Tick(void) __attribute__((signal, naked));
void IST_Tick(void)
{
        /*
           Save execution context
           Implement clock tick
           Restore new execution context
        */
    asm volatile ("reti");
}
```

AVR Solution:

```
ISR(TIMER1_COMPA_vect, ISR_NAKED) {
```

Saving the Context

```
asm volatile (
    "push r0 \n\t"
    "in r0, __SREG__ \n\t" //status register moved into R0 to be saved
    "cli \n\t" //dis. inter. (when called outside ISR, cop. sched.)
    "push r0 \n\t"
    "push r1 \n\t"
    "clr r1 \n\t"      //compiler assumes that r1 is 0
    "push r2 \n\t"
    "push r3 \n\t"
        ....
    "push r31 \n\t"    // saving task execution context finished
    "lds r26, currentTCB \n\t" // load address to which SP is saved
    "lds r27, currentTCB + 1 \n\t"
    "in r0, __SP_L__ \n\t" // save stack pointer
    "st x+, r0 \n\t"
    "in r0, __SP_H__ \n\t"
    "st x, r0 \n\t"
);
```

- program counter automatically pushed on stack
- currentTCB, currentTCB + 1: address from where task‘s stack pointer can be retrieved
- st x+,r0 stores r0 into location pointed to by X register (16 bits), increments X register

Restoring the Context

```
asm volatile (
    "lds r26, currentTCB \n\t" // currentTCB holds address of SP
    "lds r27, currentTCB + 1 \n\t"
    "ld r28, x+ \n\t" //load register 28 with data pointed at by X            //(r26,r27)
    increment register pair afterwards
    "out __SP_L__, r28 \n\t" // restore stack pointer
    "ld r29, x+ \n\t"
    "out __SP_H__, r29 \n\t"  // write contents of r29 to __SP_H__
    "pop r31 \n\t"          // pop registers
    "pop r30 \n\t"
    ....
    "pop r1 \n\t"
    "pop r0 \n\t"      // restore status register before register r0
    "out __SREG__, r0 \n\t"
    "pop r0 \n\t"
);
```

2

Multitasking Operating Systems

Overview

- Other services to be added to scheduler: timing, input/output (I/O) related features, semaphores
  - Example: Generating a time delay to lock out a task from running for some time, or waiting for an input
  - Consequence: Scheduler gets more complex
- Operating Systems (OS)
  - Subsumes scheduler and dispatcher
  - Provides abstraction of hardware
    - If OS runs on different hardware, application programmers do not need to know details of specific hardware
  - Goal: Reduction of ad hoc programming that must be done in every application
    - No need to reinventing the wheel in each application

Preemptive Prioritized Multitasking OS

- Tasks are assigned priorities (high, medium, low, …), and register with OS what events they are interested in
- Task is ready to run when one of the events it has registered for takes place
- Upon an event corresponding task is marked as ready to run
- If there is a task t_{hp} that is ready to run with higher priority that currently executing task t_r, then task t_r is preempted and control is transferred to t_{hp}
- If running task runs out of things to do it yields voluntarily

Preemptive Prioritized Multitasking OS

- Events are generated by ISRs
  - E.g. by I/O drivers supplied by OS
- ISRs can be part of OS or user supplied
- Events are stored up in FIFO queues, so if an event is not handled immediately it will not get lost
- Preemptive, prioritized systems impose a larger execution time and memory overhead than cooperative systems
  - Saving and restoring the context consumes quite an amount of memory and processor time

Summary

- Superloops
  - Perfectly fine for multitasking applications of moderate complexity
  - Not an OS, no built in services
- Round robin scheduler
  - Powerful mechanism for structuring a program, provides services such as timers, input waits etc., may result in long delays for particular events
- Event Driven Loop
  - Order of tasks is determined dynamically
  - Difficult to analyze due to asynchronous events
- Cooperative multitasking OS
  - Simple, reliable, and provide a lot of useful functionality
  - OK for single programmer, but demands strict programming discipline
- Preemptive prioritized OS
  - Generally considered best for large programs
  - Less demanding of discipline from programmers than cooperative systems
  - Causes overhead

3

Concurrent Task Model

Overview

- In a multitasking OS several copies of a task may run simultaneously or at different times
- OS manages tasks
- Usage scenarios for simultaneously running tasks
  - Multiple rates: multimedia, automotive
  - Asynchronous input: user interfaces, communication systems
- Concurrent systems consist of a set of tasks

Concurrent Task Model

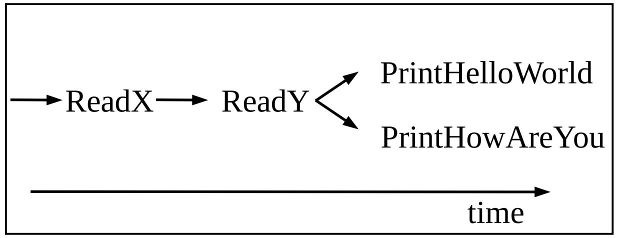
- Difficult to write concurrent system using sequential program model (user has to program interleaving actions)
- **Concurrent Task Model**
  Describes functionality of system in terms of concurrently executing subtasks disregarding interleaving aspects
- Concurrent task model easier
  - Separate sequential programs (e.g. C function for each task)
  - Tasks communicate with each other (e.g. via shared variables)
- Systems which are inherently multitasking are easier to describe with concurrent task model

Concurrent Task Model

```
ConcurrentTaskExample() {
  x = ReadX();
  y = ReadY();
  Call concurrently:
    PrintHelloWorld(x) and
    PrintHowAreYou(y);
}
PrintHelloWorld(x) {
  while(1) {
    print "Hello world";
    delay(x);
  }
}
PrintHowAreYou(y) {
  while(1) {
    print "How are you?";
    delay(y);
  }
}
```

Simple concurrent task example

- Simple example:
  - Read two integral numbers *X* and *Y*
  - Display "Hello world." every *X* seconds
  - Display "How are you?" every *Y* seconds
- More effort would be required with sequential program or state machine model
- Even better
  - OS takes care of timing!



Subroutine execution over time

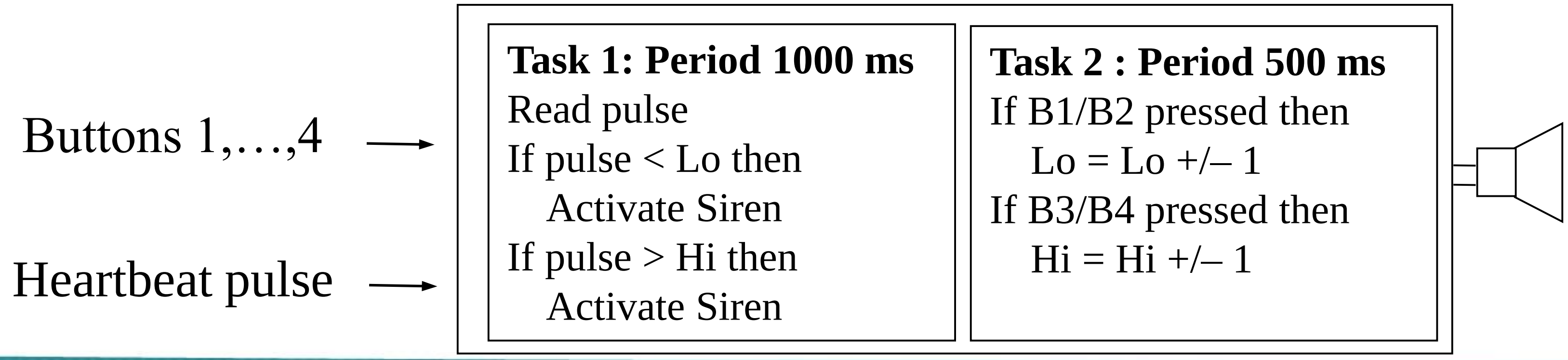
```
Enter X: 1
Enter Y: 2
Hello world.  (Time = 1 s)
Hello world.  (Time = 2 s)
How are you?  (Time = 2 s)
Hello world.  (Time = 3 s)
How are you?  (Time = 4 s)
Hello world.  (Time = 4 s)
...
```

Sample input and output

- How to implement *Call concurrently*?

Example I: Heartbeat Monitoring System

- Task 1:
  - Once per second sample input pulse and compute heartbeat of patient, signal if value is outside thresholds `Lo` and `Hi`
- Task 2:
  - Twice a second check buttons: if one of four buttons is pressed, increment/decrement `Lo` resp. `Hi`
- Tasks are independent, but access common data (`Lo`, `Hi`)
- OS schedules tasks according to periods, user not responsible for timing and interleaving the code



Example II: Set-top Box

- Task 1:
  - Receive signal from antenna, decompose into compressed video and audio
- Tasks 2 & 3:
  - Decode compressed video resp. audio
- Tasks are independent, but access common data
- OS schedules tasks (blocking and unblocking)

Input Signal →

Task 1: Period given by video format

Read Signal
Separate Audio/Video
Send Audio to Task 2
Send Video to Task 3

Task 2: Blocking operation
Wait on Task 1
Decode/output Audio

→ Audio

Task 3: Blocking operation
Wait on Task 1
Decode/output Video

→ Video

Basic Operations on Tasks

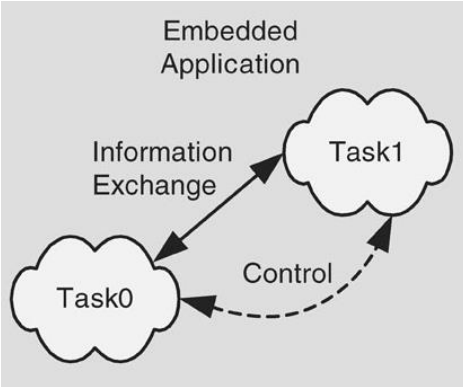
- Create and terminate
  - Create is like a procedure call but caller doesn’t wait (block)
    - Created task can itself create new tasks
    - In HelloWorld/HowAreYou example, two tasks are created
  - Terminate kills a task, destroying all data
- Suspend and resume
  - Suspend puts task on hold, saving state for later execution
  - Resume starts the task again where it left off
- Join
  - Task suspends until a particular child task finishes execution
- Wait (block)
  - Task waits for condition to be fulfilled by other task (avoiding busy waiting)

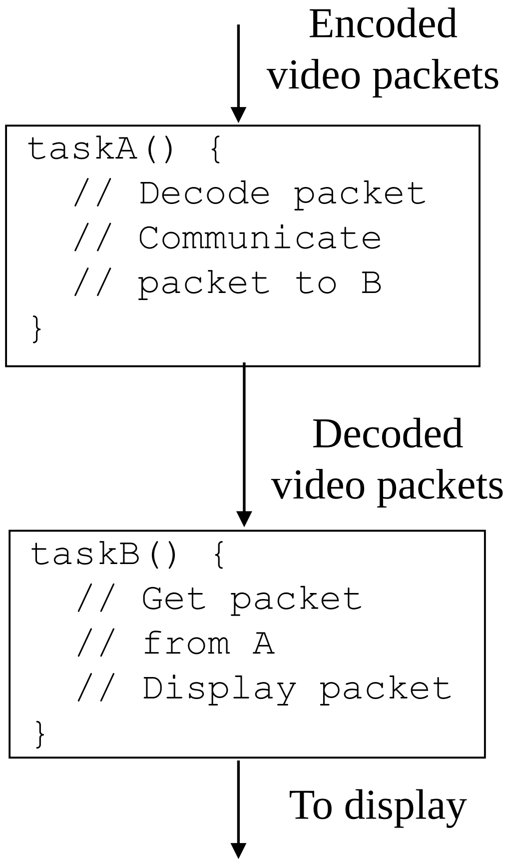
4

Communication Among Tasks

4. Communication Among Tasks

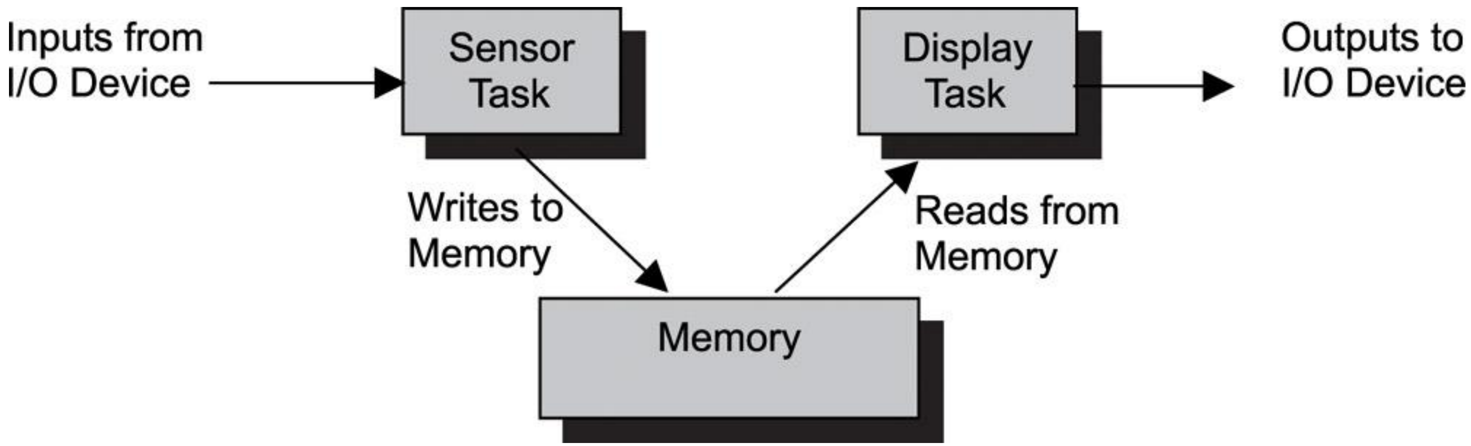
- Tasks need to communicate data and signals to solve their computation problem
  - Tasks that don't communicate are just independent programs solving separate problems
- Basic pattern: producer/consumer
  - Task A produces data and B consumes it
  - Example: A decodes video packets, B displays decoded packets on a screen
- How do we achieve this communication?
  - Two basic methods
    - Shared memory
    - Message passing
  - What to do if tasks run on variable speeds?
    - Buffers





Shared Memory

- Tasks read/write shared variables
  - No time overhead, easy to implement
  - Example with two tasks:
    - Sensor task periodically receives data from a sensor and writes data to shared memory
    - Display task periodically reads from shared memory and sends data to display
  - Access by multiple tasks **must** be synchronized to maintain the integrity of shared data



Shared Memory - Example

- Producer/consumer with preemptive scheduler
  - Shared variables: *buffer*[*N*], *count*
    - *count* = # of valid data items in *buffer*
  - *taskA* produces data items & stores them in *buffer*
    - If *buffer* is full, then taskA must wait
  - *taskB* consumes data items from *buffer*
    - If *buffer* is empty, then taskB must wait
  - Race condition:
    Error when both tasks try to update *count* concurrently (lines 10 and 19) and the following execution sequence occurs: (let count = 3)
    - *A* loads *count* (3) from memory into register R1 (R1 = 3)
    - *A* increments R1 (R1 = 4)
    - *B* loads *count* (3) from memory into register R2 (R2 = 3)
    - *B* decrements R2 (R2 = 2)
    - *A* stores R1 back to *count* in memory (*count* = 4)
    - *B* stores R2 back to *count* in memory (*count* = 2)

    count now has incorrect value of 2
  - No busy waiting in lines 7, 16

```
01: data_type buffer[N]; // shared
02: int count = 0;  // shared

03: void taskA() {
04:   int i = 0;//next slot to store
05:   while(1) {
06:     produce(&data);
07:     while (count == N)
          wait_task;
08:     buffer[i] = data;
09:     i = (i + 1) % N;
10:     count = count + 1;
11:   }
12: }

13: void taskB() {
14:   int i = 0;//next slot to read
15:   while(1) {
16:     while (count == 0)
          wait_task;
17:     data = buffer[i];
18:     i = (i + 1) % N;
19:     count = count - 1;
20:     consume(&data);
21:   }
22: }

23: void main() {
24:   create_task(taskA);
25:   create_task(taskB);
26: }
```

Message Passing

- Message passing
  - Data explicitly sent from one task to another
    - Sending task performs special operation *send to send data*
    - Receiving task performs special operation *receive* to receive the data
    - Both operations must explicitly specify which task it is sending to or receiving from
    - Receive is blocking, send is not blocking
    - Blocked task waits for another task to send data
  - Safer model, but less flexible

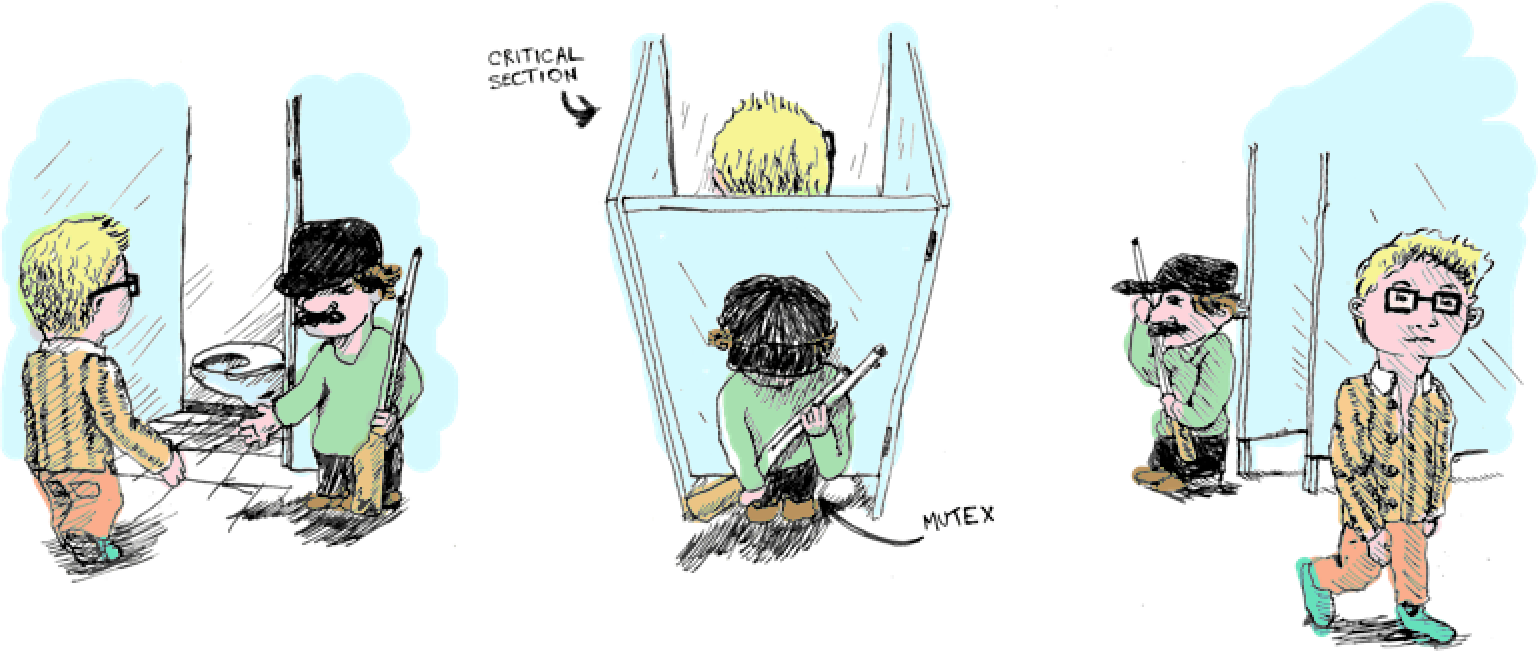
```
taskA() {
  while(1) {
    produce(&data)
    send(B, &data);
    /* do something */
    receive(B, &data);
    consume(&data);
  }
}
```

```
taskB() {
  while(1) {
    receive(A, &data);
    transform(&data)
    send(A, &data);
    /* do something */
  }
}
```

Back to Shared Memory: Mutual Exclusion

- Certain sections of code should not be performed concurrently
  - **Critical section**:
    Section of code where simultaneous updates, by multiple tasks to a shared memory location, can occur
- When a task enters the critical section, all other tasks must be locked out until it leaves the critical section
- Solution with Mutex (requires hardware support)
  - A shared object used for locking and unlocking segment of shared data
  - Disallows read/write access to memory it guards
  - Multiple tasks can perform lock operation simultaneously, but only **one** task will acquire lock
  - All other tasks trying to obtain lock are put in blocked state until unlock operation performed by acquiring task when it exits critical section
  - These tasks will then compete for lock again

Mutex



Implementing a Lock

- Shared Boolean variable: `locked`
- Value indicates whether access to critical section is currently allowed
- Access to variable `locked` using two functions

```
void lock() { //blocking function
    ......
}
void unlock() {
    ......
}
```

- Usage: Placement of code of critical section

```
lock();
    // code of critical section
unlock();
```

Implementing a Lock

- `locked`: pointer to lock variable

```
void lock() {
    while (*locked);
    *locked = true;
}
```

- Atomic operations on some processors
  - test-and-set, fetch-and-add, compare-and-swap
- **Logical** implementation of test-and-set (in reality one machine instruction)

```
boolean testAndSet(boolean * target) {
    boolean rv = * target;
    * target = true;
    return rv;
}
```

- Usage

```
void lock() {
    while (testAndSet(locked)); //busy waiting
}
void unlock() {
    locked = false;
}
```

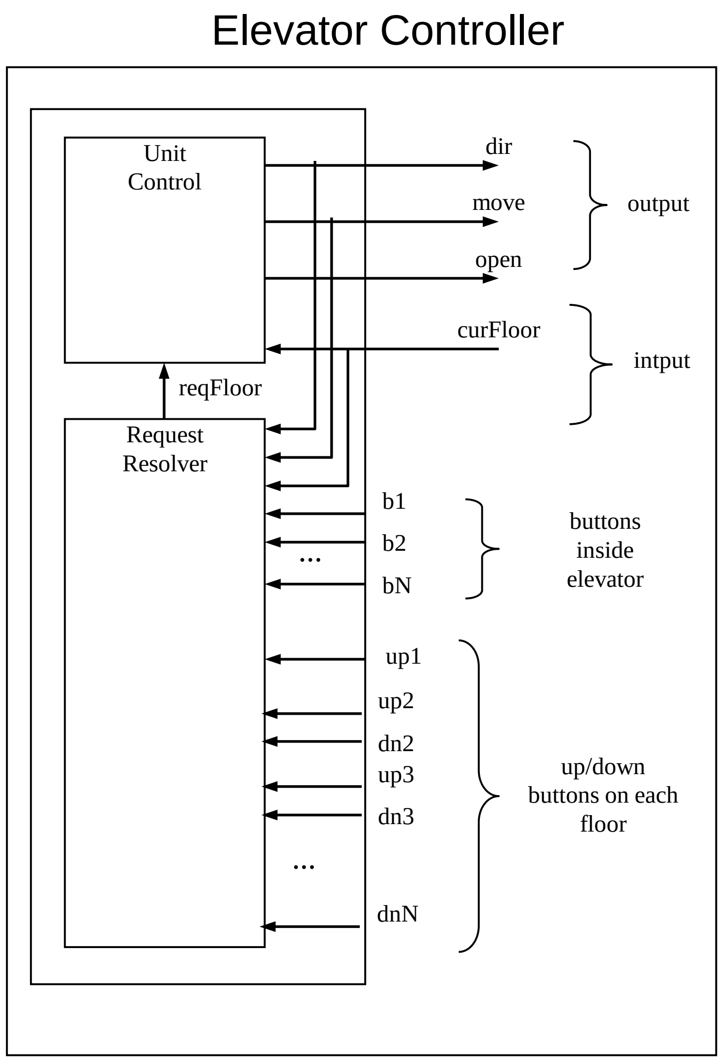
Consumer-Producer Problem with Mutex

- *mutex* is used to ensure that critical sections are executed in mutual exclusion of each other
- Same execution sequence as before:
  - *A*/*B* execute *lock* operation on *count_mutex*
  - Either *A* **or** *B* will acquire *lock*
    - Say *B* acquires it
    - *A* will be put in blocked state
  - *B* loads *count* (3) from memory into register R2 (R2 = 3)
  - *B* decrements R2 (R2 = 2)
  - *B* stores R2 back to *count* in memory (2)
  - *B* executes *unlock* operation
    - *A* is placed in runnable state again and acquires lock
  - *A* loads *count* (2) from memory into register R1 (R1 = 2)
  - ……
- *Count* now has correct value of 3

```
data_type buffer[N];
int count = 0;
mutex count_mutex;

void taskA() {
  int i = 0;
  while(1) {
    produce(&data);
    while (count == N)
        wait_task;
    buffer[i] = data;
    i = (i + 1) % N;
    count_mutex.lock();
    count = count + 1;
    count_mutex.unlock();
  }
}

void taskB() {
  int i = 0;
  while(1) {
    while (count == 0)
        wait_task;
    data = buffer[i];
    i = (i + 1) % N;
    count_mutex.lock();
    count = count - 1;
    count_mutex.unlock();
    consume(&data);
  }
}
```

Example: Task Communication

- Elevator controller
- Unit control task:
  reads next requested floor from queue and moves to requested floor
- Request resolver task:
  appends new requests to queue
  → Consumer-Producer problem
- Realization:
  - Shared variable for queue
  - Mutex for controlling access to queue



Mutual Exclusion with AVR

- `#define ATOMIC_BLOCK (type)`
  - Creates block of code guaranteed to be executed atomically
  - Upon entering block Global Interrupt Status flag in SREG is disabled
  - State of SREG upon exit is controlled by parameter `type`
    - `#define ATOMIC_FORCEON`
      - causes ATOMIC_BLOCK to force state of SREG on exit, enabling the Global Interrupt Status flag bit (i.e. previous value of SREG register is not need saved at start of block)
    - `#define ATOMIC_RESTORESTATE`
      - causes ATOMIC_BLOCK to restore previous state of SREG register, saved before Global Interrupt Status flag bit was disabled

Deadlocks in Concurrent Programming

- Deadlock: Condition where two tasks are blocked waiting for other to unlock critical sections
  - Both tasks are then in blocked state and
  - cannot execute unlock operation and wait forever
- Example: Two different critical sections that can be accessed simultaneously
  - Two locks needed (mutex1, mutex2)
  - Following execution sequence produces deadlock
    - *A* executes lock operation on *mutex1* (and acquires it)
    - *B* executes lock operation on *mutex2* (and acquires it)
    - *A*/*B* both execute in critical sections 1 and 2, resp.
    - *A* executes lock operation on *mutex2*
      - *A* blocked until *B* unlocks *mutex2*
    - *B* executes lock operation on *mutex1*
      - *B* blocked until *A* unlocks *mutex1*
    - DEADLOCK!
- Possible deadlock elimination protocols
  - Acquire all needed locks in one step
  - Number locks and acquire them in increasing order

```
mutex mutex1, mutex2;

void taskA() {
  while(1) {
    …
    mutex1.lock();
    /* critical section 1 */
    mutex2.lock();
    /* critical section 2 */
    mutex2.unlock();
    /* critical section 1 */
    mutex1.unlock();
  }
}

void taskB() {
  while(1) {
    …
    mutex2.lock();
    /* critical section 2 */
    mutex1.lock();
    /* critical section 1 */
    mutex1.unlock();
    /* critical section 2 */
    mutex2.unlock();
  }
}
```

Synchronization Among Tasks

- Concurrently running tasks must synchronize their execution When a task must wait for
  - another task to compute some value
  - reach a known point in their execution
  - signal some condition
- Recall producer-consumer problem
  - *taskA* must wait if *buffer* is full
  - *taskB* must wait if *buffer* is empty
- Busy-waiting
  - Tasks execute loops instead of being blocked
  - CPU time is wasted!
- More efficient methods avoid busy waiting
  - Join operation, and blocking send and receive discussed earlier
    - Both block task, no wastage of CPU time
  - Condition variables and monitors avoid busy waiting

SES

Chapter 6: Operating Systems for Embedded Systems

Prof. Dr.-Ing. Bernd-Christian Renner

Institute smartPORT
Hamburg University of Technology

TUHH