Real **T**ime **O**perating **S**ystems
-**RTOS**-

Kaiserslautern University of Technology
June 2006

Ramon Serna Oliver
serna_oliver@eit.uni-kl.de

Outline

- Basic concepts
- Real Time Operating Systems
- RTOS Design considerations
- Tasks and scheduler in RTOS
- Tasks communication and synchronization
- Example
- POSIX Standard
- Conclusions and references

Outline

- **Basic concepts**
- Real Time Operating Systems
- RTOS Design considerations
- Tasks and scheduler in RTOS
- Tasks communication and synchronization
- Example
- POSIX Standard
- Conclusions and references

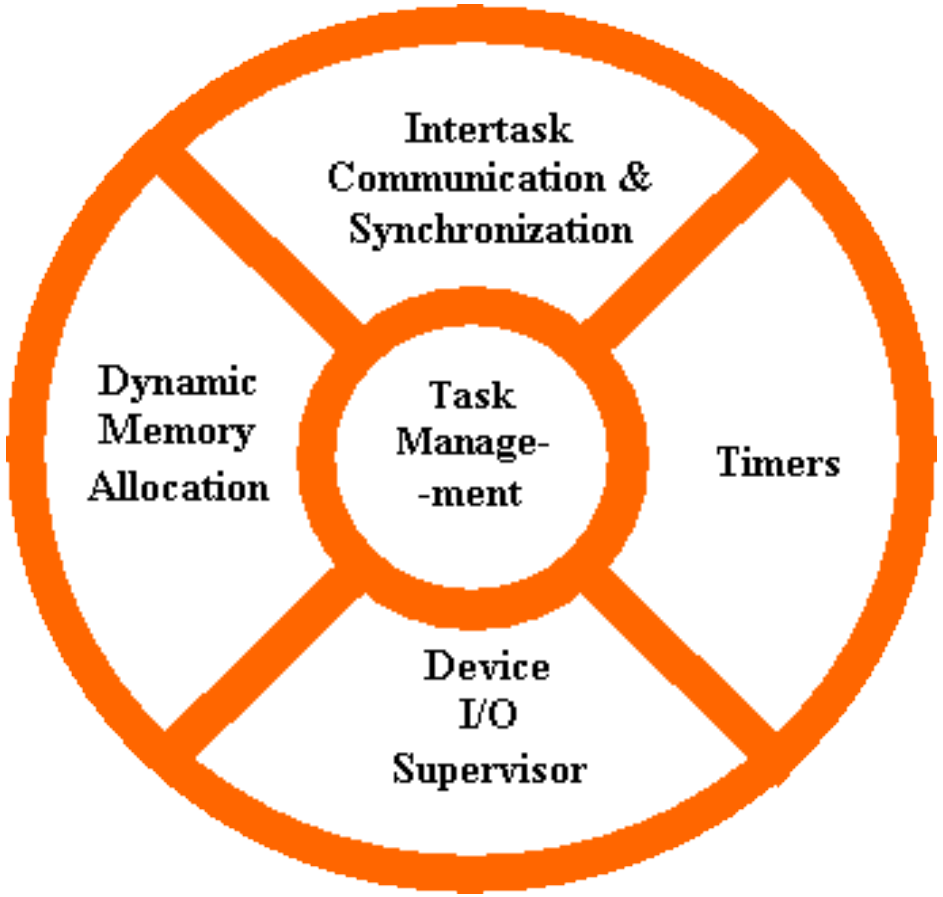
Basic concepts

- What is the OS?
- Basic structure of the OS
- OS classification

What is the OS?

- Collection of system calls (functions)
  - Provide a set of basic services to interact with the hardware
- The core of the OS is the Kernel
  - Typically a library or set of libraries
  - Tends to be small and highly optimized
  - Might be (or not) other high level services on top
    - graphic interface, desktop environment, i/o, user interfaces, network, etc...

Basic structure of the OS



- An operating system provides:
  - Resource allocation
  - Task management, scheduling and interaction
  - Hardware abstraction
  - I/O interface
  - Time control
  - Error handling

OS classification

- Depending on...
  - response time: batch / interactive / real-time
  - users: multi / single user
  - execution mode: multi / single task
  - others: embedded, portable, standard compliant (i.e. POSIX).
- Real OS are a mixture of the above.

Outline

- Basic concepts
- **Real Time Operating Systems**
- RTOS Design considerations
- Tasks and scheduler in RTOS
- Tasks communication and synchronization
- Example
- POSIX Standard
- Conclusions and references

Real Time Operating Systems - RTOS

- Main features
- Other features

RTOS: Main features

- Same basic structure than a regular OS, but...
- In addition it provides mechanisms to allow real time scheduling of tasks:
  - **Deterministic** behavior of all system calls (!)
    - All operations (*system calls*) must be **time bounded**
      - No memory swap (*virtual memory*) is typically allowed
      - “Special” hw interaction. Ad-hoc drivers (!)
    - Interrupt service routines (ISR) **time bounded**

RTOS: Other features

- Task priority policy
  - Can be dynamic or static
- Inter-task communication and synchronization
  - semaphores, mailboxes, message queues, buffers, monitors, ...
- Hardware access is done at a lower level
  - faster, but less “comfortable” to the user
- Modular structure

  - Unused modules can be discarded to save space/resources

Outline

- Basic concepts
- Real Time Operating Systems
- **RTOS Design considerations**
- Tasks and scheduler in RTOS
- Tasks communication and synchronization
- Example
- POSIX Standard
- Conclusions and references

RTOS Design considerations

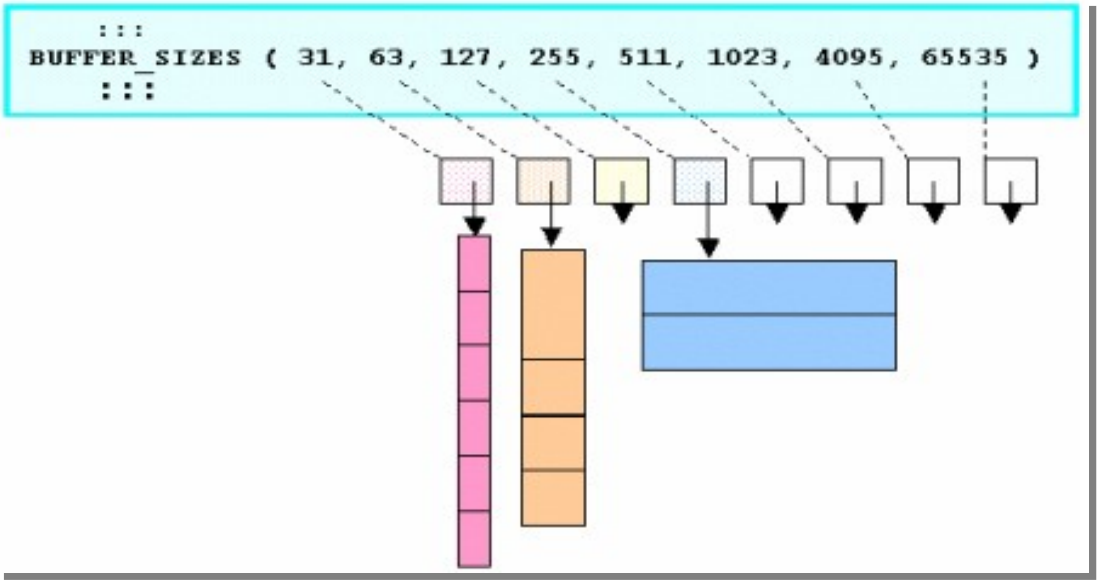
- Memory management
- I/O Management
- Time handling
- Limited resources considerations

Memory Management

- Common memory handling is unpredictable
  - Memory allocation time increases depending on
    - block size
    - status of the memory (“clean or dirty”)
      - fragmentation
    - swap / virtual memory (depends on HD)
  - Solution
    - 1) No memory handling at all
    - 2) Limited memory management

Memory Management (II)

- Example of RTOS memory manager:
  - Only certain blocs of fixed size of dynamic memory can be required by tasks.
  - Blocks are stored in a “free buffer queue”



Memory Management (III)

- Advantages:
  - No fragmentation
    - A block is always taken and used as an unit
  - No need of garbage collection (!)
  - Easily implemented as constant functions (deterministic behavior!)
    - get_mem() and set_mem() vs malloc() and free()
- Disadvantages:
  - Limited choice of chunks size
  - Suboptimal usage of memory resources

I/O Management

- Unpredictable hw behavior:
  - No control over the device behavior
    - ie: disc segmentation, hw failure, network collision, random delays (CSMA/CD), ...
  - I/O functions are typically non reentrant
    - ie: printf() is not appropriated for RT applications
    - New reentrant functions provided, ie:

```
void safe_printf(char *txt) {
    get_semaphore(s);
    printf(txt);
    release_semaphore(s);
}
```

I/O Management (II)

- CPU & RAM are controllable
- Other devices are commonly unpredictable
  - i.e. HD access is non deterministic
    - Depends on header position, latency, spinning speed,...
  - Manufacturers drivers are not Real Time compliant
    - Development of ad-hoc drivers (!)
    - Assembler and low level code = platform dependent

Time handling

- Time control is a main thing
  - Task scheduling
    - deadlines (!)
    - delay() & delay_until() functions
  - Time outs
- Time resolution
  - depends on hw
  - might be adjustable by the user
    - >resolution → >overhead (!)

Limited resources

- RTOS commonly oriented to Embedded systems
  - Little resources available
    - micro controllers
    - low power CPUs
    - ...
  - Little space to add cool stuff
    - Size restrictions
    - Limited power consumption
    - ...

Outline

- Basic concepts
- Real Time Operating Systems
- RTOS Design considerations
- **Tasks and scheduler in RTOS**
- Tasks communication and synchronization
- Example
- POSIX Standard
- Conclusions and references

Tasks and scheduler in RTOS

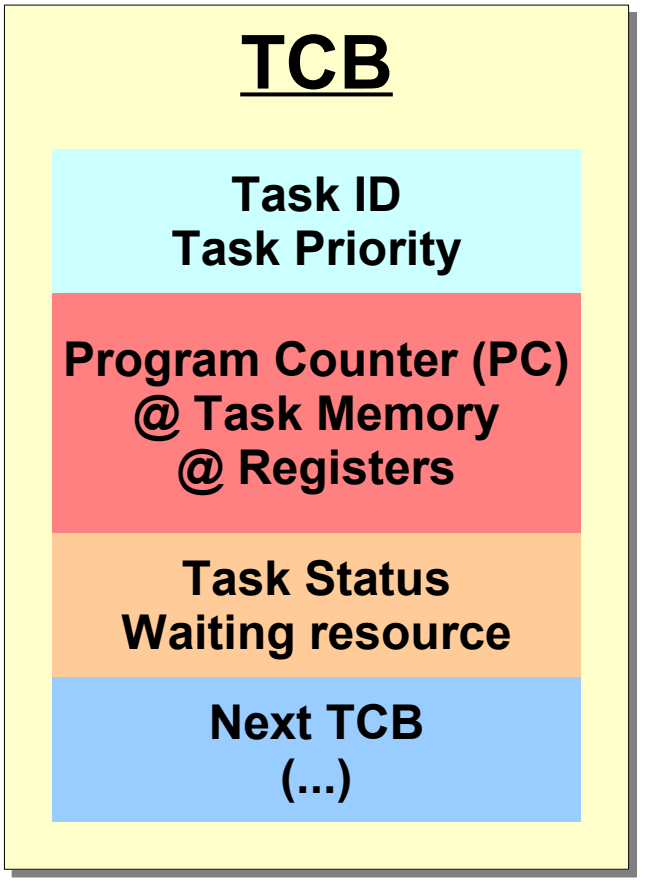
- What is a task for us (users/programmers)?
- What is a task for the OS?
- Task states
- Scheduler

What is a task for us?

- A task is a piece of executable code
  - a function (procedure)
  - a program
  - a kernel module
  - ...
- A program consists of one or more tasks
  - Tasks run concurrently
    - Real concurrence in multi-processor system
    - Pseudo concurrence in single-processor systems

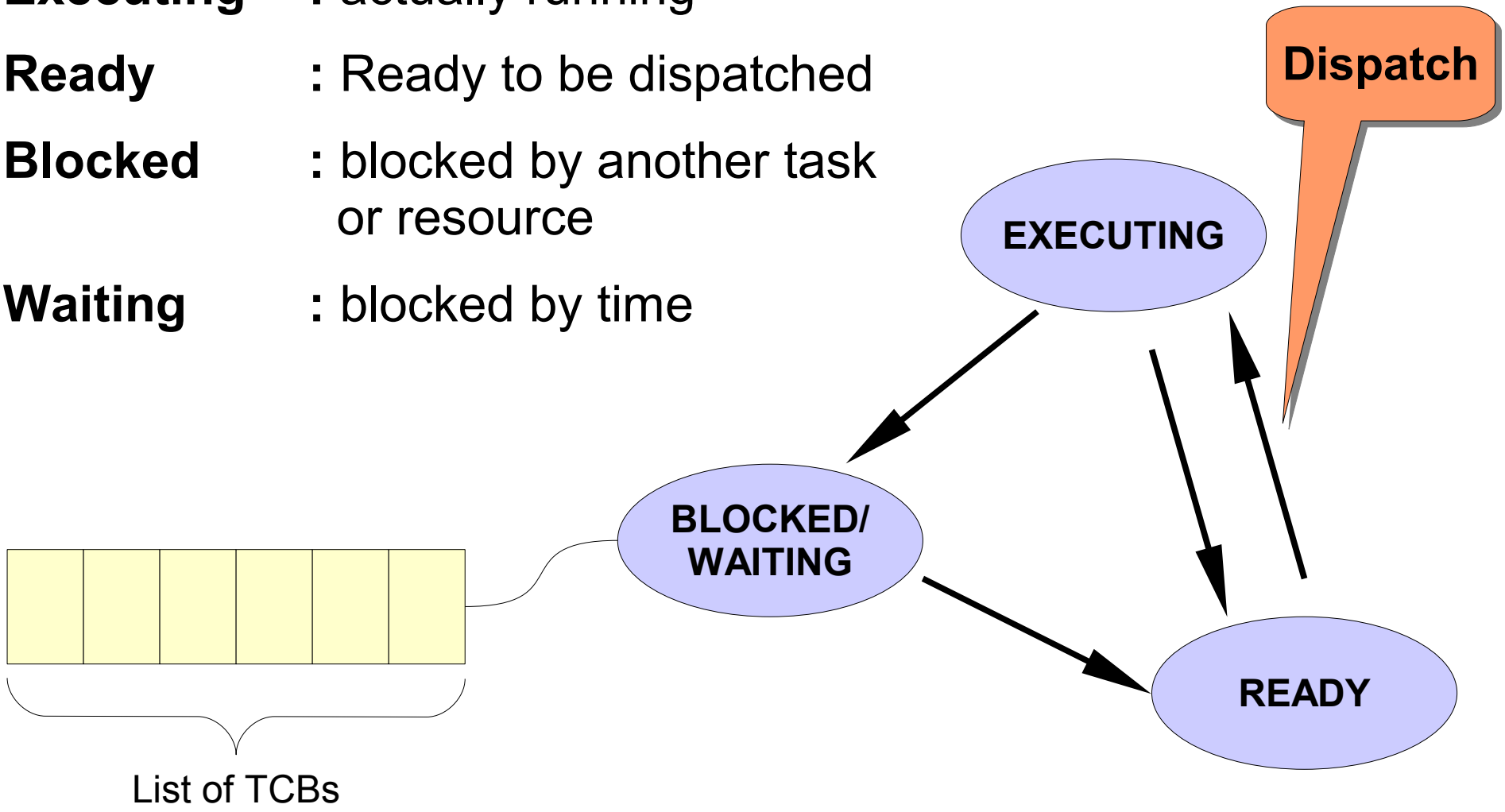
What is a task for the OS?

- Task Control Block (TCB)
  - Task ID and Task Priority
  - Copy of last execution context
    - CPU registers, stack, ...
  - Task status
  - Other relevant information
- Create task = create TCB
- Destroy task = delete TCB



Task States

- **Executing** : actually running
- **Ready** : Ready to be dispatched
- **Blocked** : blocked by another task or resource
- **Waiting** : blocked by time



Scheduler

- Scheduler
  - Implement a scheduling policy
    - select next task to be executed
    - maintain queues (order by?)
    - maintain priorities (static? dynamic?)
  - Time triggered
    - Kernel executes periodically
  - Event triggered
    - Kernel executes only when “something” happens

Outline

- Basic concepts
- Real Time Operating Systems
- RTOS Design considerations
- Tasks and scheduler in RTOS
- **Tasks communication and synchronization**
- Example
- POSIX Standard
- Conclusions and references

Communication & Synchronization

- Needs of communication and synchronization
- Communication mechanisms
- Synchronization mechanisms

Tasks communication and Synchronization

- Tasks need to communicate
  - sharing data
    - i.e. producer / consumer
    - mutual exclusion (mutex)
    -
- Tasks need to synchronize
  - “meeting points”
    - i.e. wait for an intermediate result
    - i.e. wait for the end of another task

Communication

- Mailboxes:
  - A task sends a message to a mailbox
    - message_send(id_mailbox, message);
      - Task blocked if mailbox full
  - A task reads a message from a mailbox
    - m= message_read(id_mailbox);
      - Task blocked until a message is available
      - Message deleted after reading

Communication (II)

- Message queues:
  - Same concept as Mailboxes but more than one message can be stored (fixed capacity)
    - Sender can send up to N messages before being blocked
    - Reader gets messages in FIFO order

Communication

- Mutex
  - Two basic **atomic** operations:
    - mutex_lock()
      - blocks execution until mutex is unlocked
      - then locks mutex
    - mutex_unlock()
      - unlock mutex
  - Only one task can enter a section protected by mutex.
  - optional protocols to avoid priority inversion

Communication

- Example:

```
mutex m;

int global_id = 0;
```

Task A

```
void taskA() {
    while (1) {
        mutex_lock(m)
        id_a = global_id;
        global_id ++;
        mutex_unlock(m);
        ...
    }
}
```

Task B

```
void taskB() {
    while (1) {
        mutex_lock(m)
        id_b = global_id;
        global_id ++;
        mutex_unlock(m);
        ...
    }
}
```

Synchronization

- Semaphores
  - Two basic operations (similar to MUTEX)
    - wait_semaphore()
      - decrement resource counter
      - block execution if counter = 0
    - signal_semaphore()
      - increment resource counter
      - if value<=0 one of the blocked tasks continues execution
  - Plus initial value 
    - number of available resources (>=0)
      - resource counter

Synchronization

- Example:

```
semaphore s =
   init_semaphore(0);
```

Task A

```
void taskA() {
    while (1) {
        ...
        produce_result();
        signal_semaphore(s);
        ...
    }
}
```

Task B

```
void taskB() {
    while (1) {
        ...
        wait_semaphore(s)
        get_result();
        ...
    }
}
```

Outline

- Basic concepts
- Real Time Operating Systems
- RTOS Design considerations
- Tasks and scheduler in RTOS
- Tasks communication and synchronization
- **Example**
- POSIX Standard
- Conclusions and references

Example

- Simple example with two tasks
- Notes

Example (I)

Task 1 (T1)

```
void task1(void *param) {
   while (1) {
      ...
      echo("task 1");
      sleep_until(time+10);
   }
}
```

Priority = 1

Period = 10

Execution time = 5

Task 2 (T2)

```
void task2(void *param) {
   while (1) {
      ...
      echo("task 2");
      sleep_until(time+50);
   }
}
```

Priority = 2

Period = 50

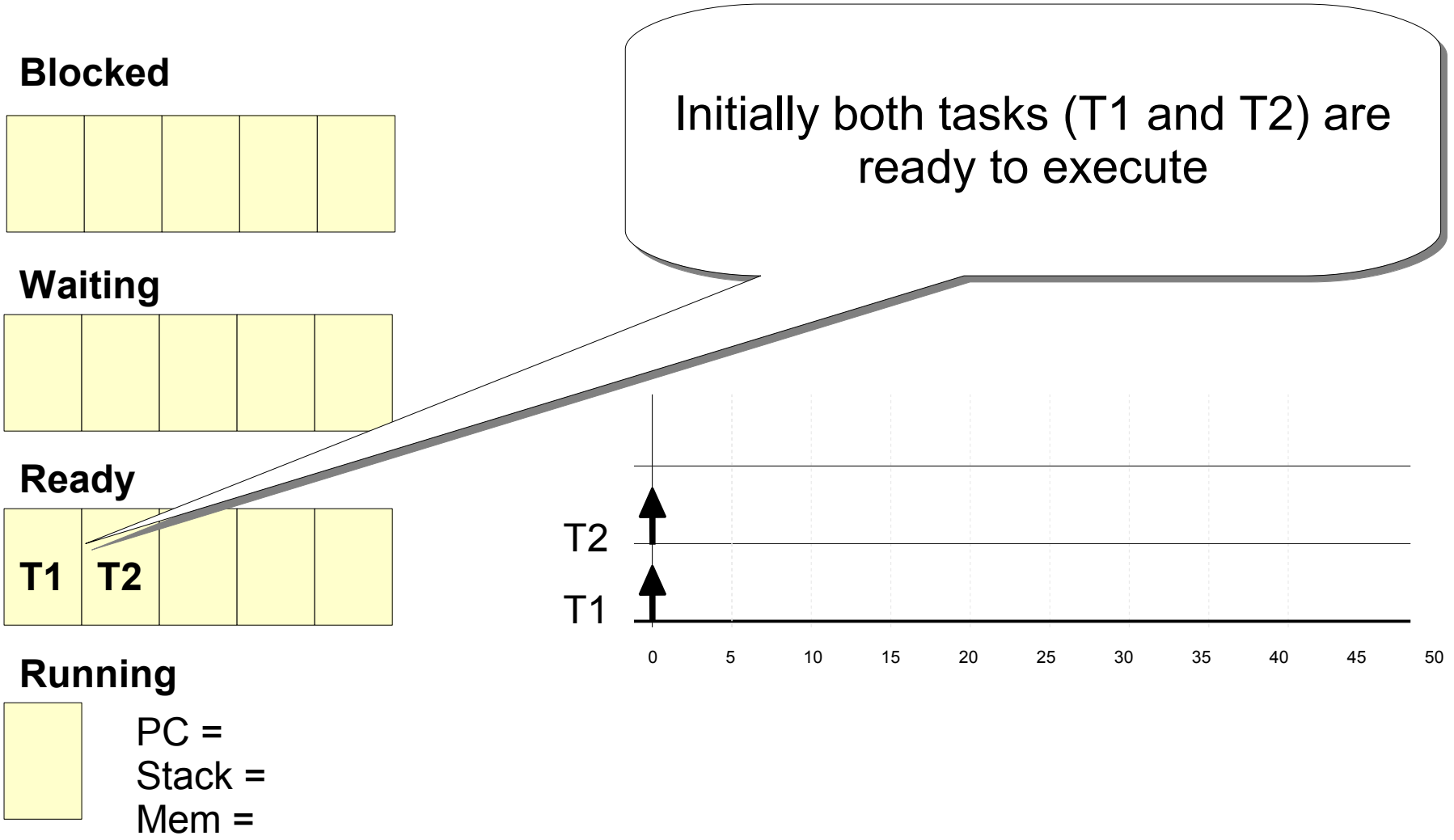
Execution time = 10

Example (II)

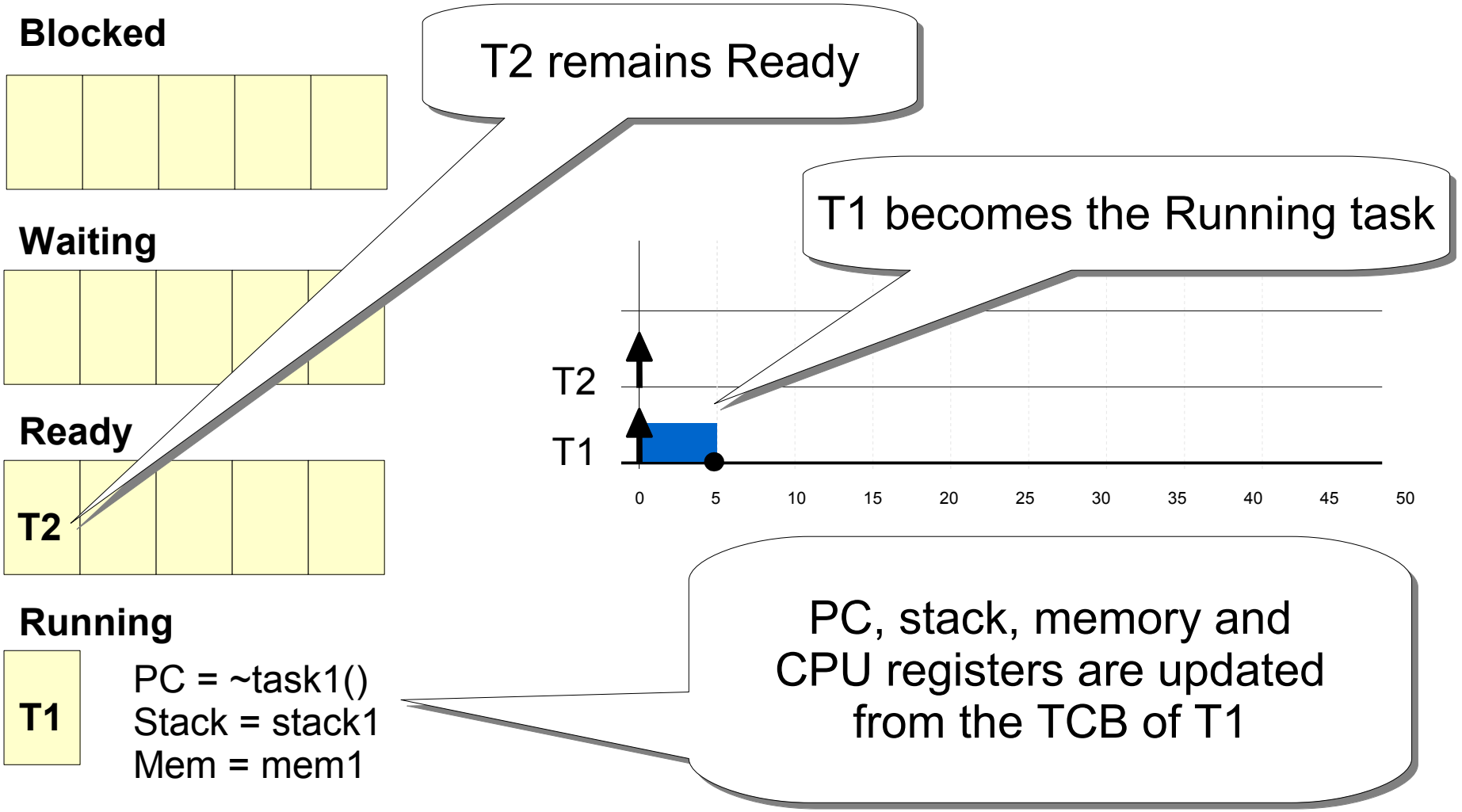
Task Creation (main)

```
void main() {

   char *mem1[100], *mem2[100];
   int t1, t2;

     /* create_task(@code, @param, @mem, priority */
   t1 = create_task(*task1, NULL, mem1, 1);
   t2 = create_task(*task2, NULL, mem2, 2);

     /* OS starts executing */
   os_start();
} /* end */
```

Example (III)


Blocked
Waiting
Ready
T1
T2
Running
PC =
Stack =
Mem =
Initially both tasks (T1 and T2) are ready to execute
T2
T1
0
5
10
15
20
25
30
35
40
45
50

Example (IV)


Blocked
Waiting
Ready
T2
Running
T1
PC = ~task1()
Stack = stack1
Mem = mem1
T2 remains Ready
T1 becomes the Running task
T2
T1
0
5
10
15
20
25
30
35
40
45
50
PC, stack, memory and
CPU registers are updated
from the TCB of T1

Example (V)

T1 reaches the end of the loop.
Sleeps until “time+10”

Blocked

Waiting

T1

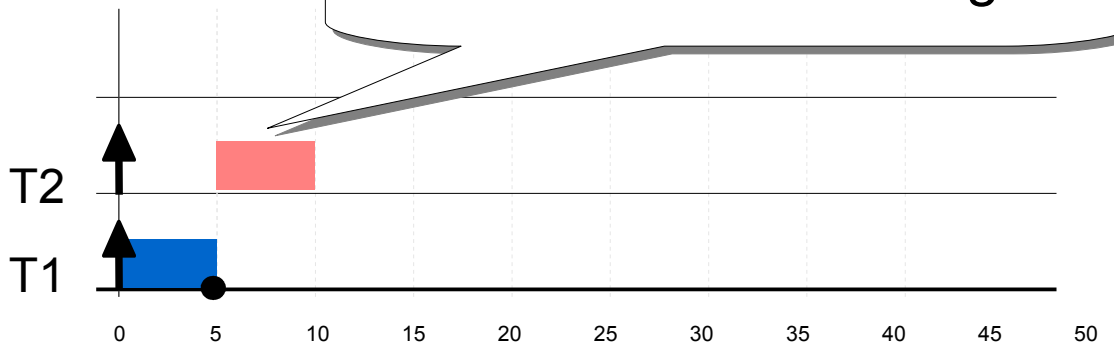
Ready

Running

T2

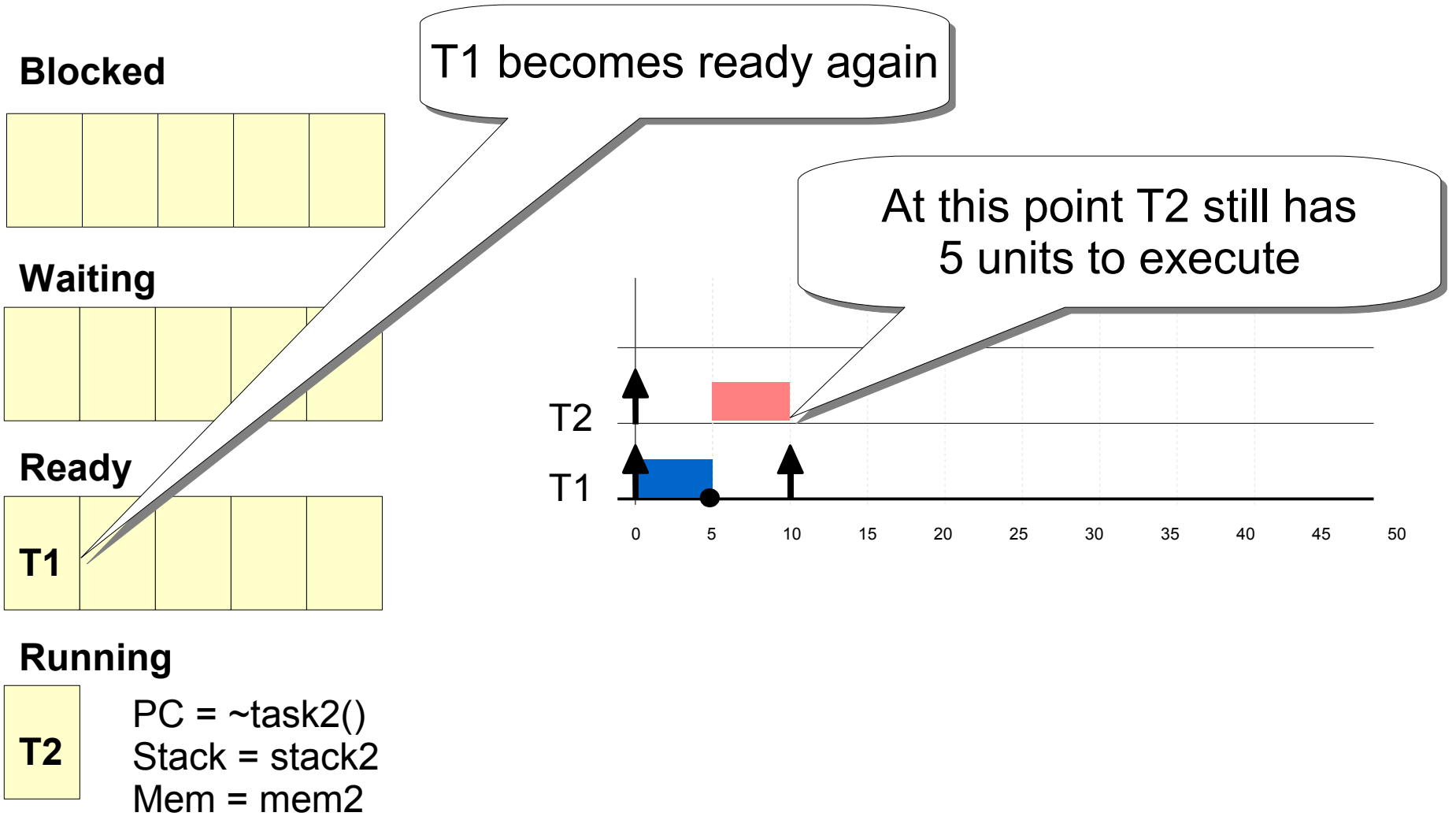
PC = ~task2()
Stack = stack2
Mem = mem2

T2 becomes the Running task



TCB of T1 is updated with PC, stack, memory and CPU registers.
PC, stack, memory and CPU registers are updated from the TCB of T2

Example (VI)


Blocked
Waiting
Ready
T1
Running
T2
PC = ~task2()
Stack = stack2
Mem = mem2
T1 becomes ready again
At this point T2 still has
5 units to execute
T2
T1
0
5
10
15
20
25
30
35
40
45
50

Example (VII)

T1 **preempts** T2 since it has higher priority

Blocked

Waiting

Ready

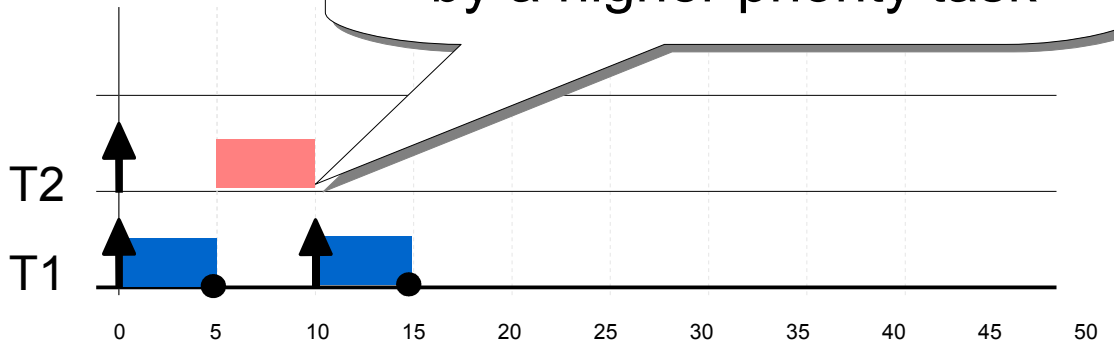
T2

Running

T1

PC = ~task1()
Stack = stack1
Mem = mem1

Execution of T2 is **preempted** by a higher priority task



TCB of T2 is updated with PC, stack, memory and CPU registers. PC, stack, memory and CPU registers are updated from the TCB of T1

Example (VIII)

T1 reaches again the end of the loop and finishes its execution

Blocked

Waiting

T1

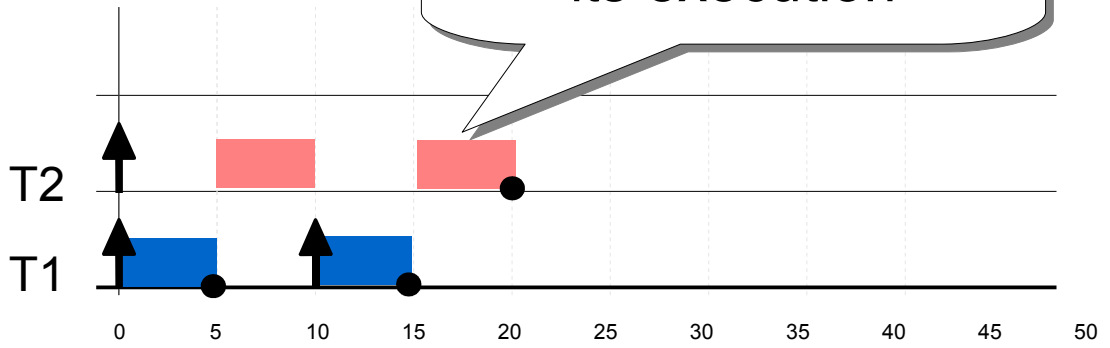
Ready

Running

T2

PC = ~task2()
Stack = stack2
Mem = mem2

T2 can now continue its execution



TCB of T2 is updated with PC, stack, memory and CPU registers. PC, stack, memory and CPU registers are updated from the TCB of T1

Example (IX)

T2 reaches the end of its loop and finishes its execution

Blocked

Waiting

T2

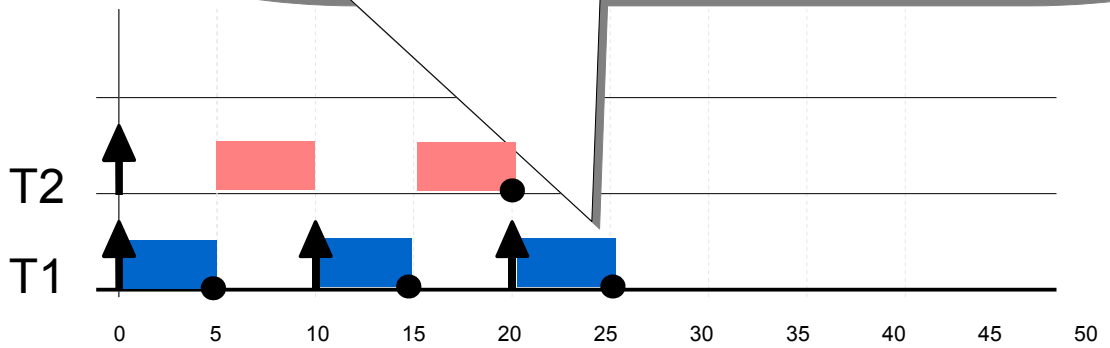
Ready

Running

T1

PC = ~task1()
Stack = stack1
Mem = mem1

T1 executes again (after moving from Waiting to Ready)



TCB of T2 is updated with PC, stack, memory and CPU registers. PC, stack, memory and CPU registers are updated from the TCB of T1

Example (X)

T1 reaches the end of
its loop and finishes its execution

No task is ready to be executed!!

Blocked

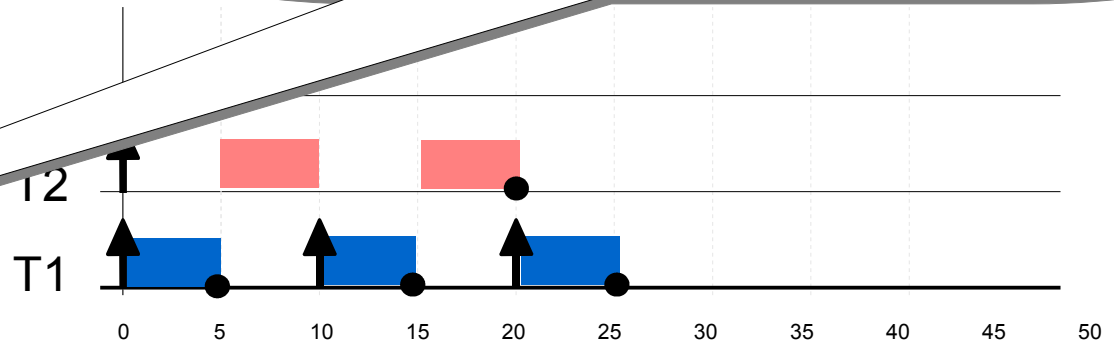
Waiting

T2 **T1**

Ready

Running

PC = ??
Stack = ??
Mem = ??



And now what

?

Example (XI)

A special task **IDLE** is scheduled with the lowest priority in order to get the CPU when no other task is ready to execute

Blocked

Waiting

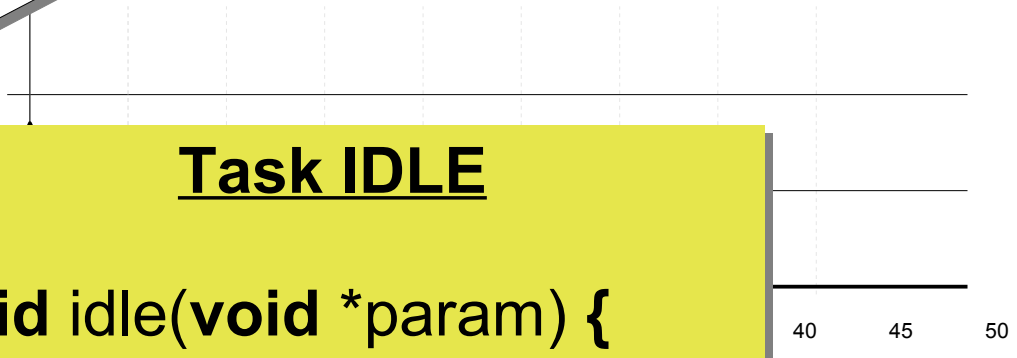
T2 **T1**

Ready

Idle

Running

PC = ??
Stack = ??
Mem = ??



Task IDLE

```
void idle(void *param) {
    while (1) { NULL }
}
```

Priority = *Lowest*
Always Ready to execute

Example (XII)

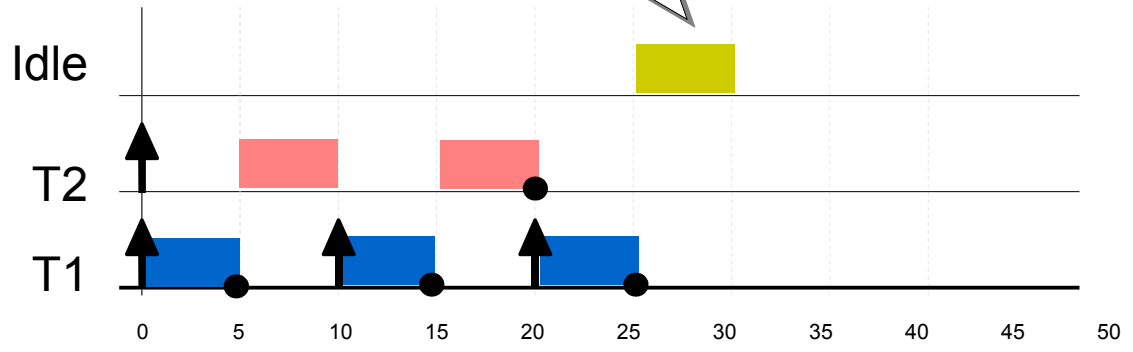
The Idle task executes until any other task is available in the Ready queue

Blocked

Waiting

T2 **T1**

Ready



Running

Idle

PC = ~idle()
Stack = stack_idle
Mem = mem_idle

Again PC, stack and memory are replaced like with any other task

Example (end)

Blocked

Waiting

...

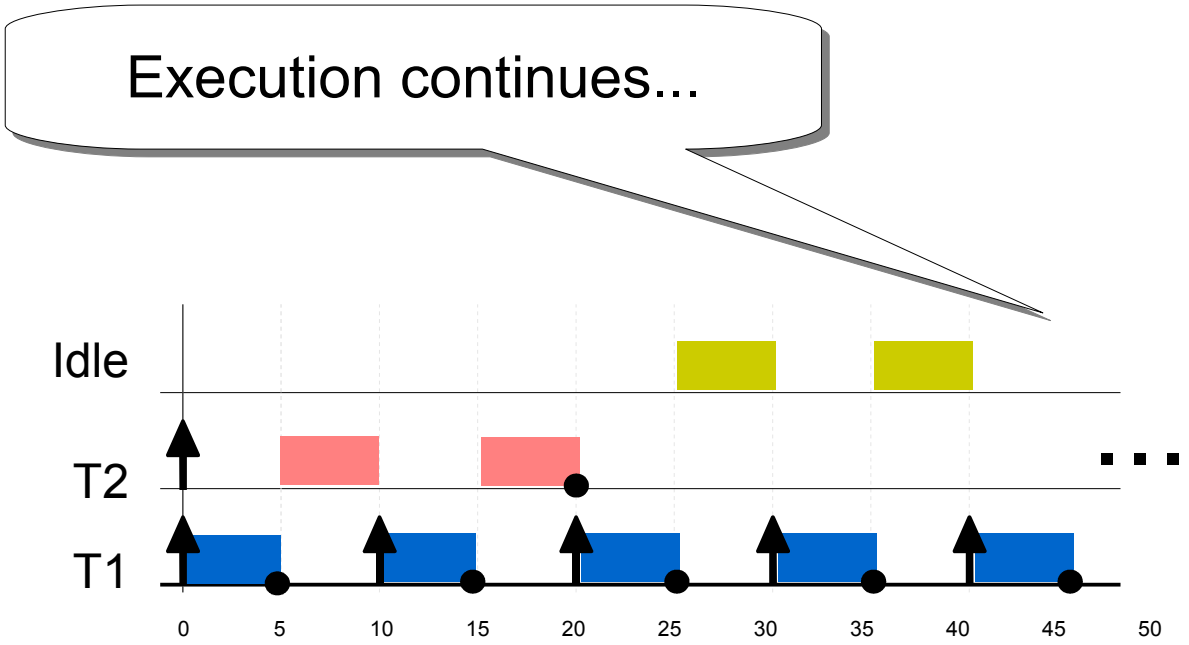
Ready

...

Running

...

PC = ...
Stack = ...
Mem = ...

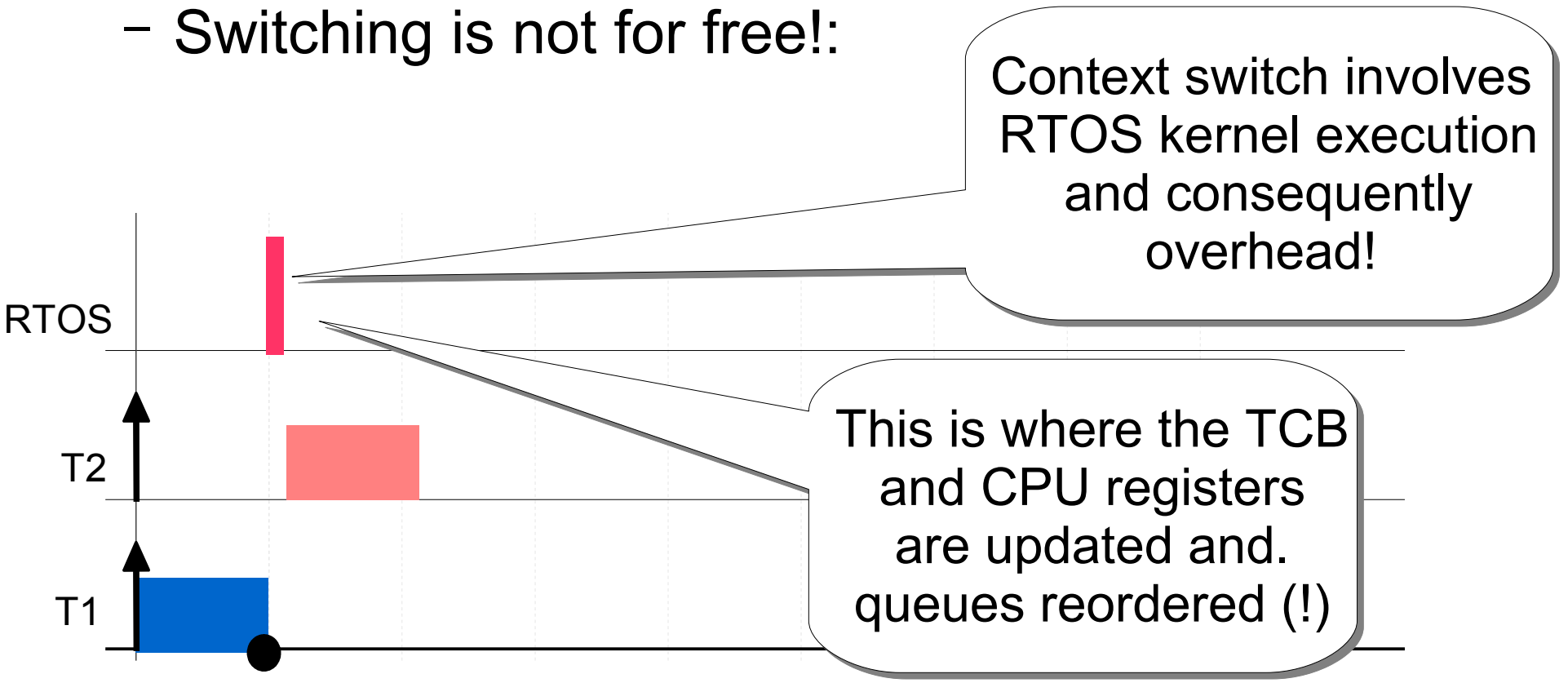


Notes from the example

- Idle task is an infinite loop that gains CPU access whenever there is “nothing to do”.
- RTOS are **preemptive** since they “kick out” the current running task if there is a higher priority task ready to execute.
- Scheduling is based on selecting the appropriated task from (ordered) queues.
- Task context is saved on its own TCB and restored in every new execution.

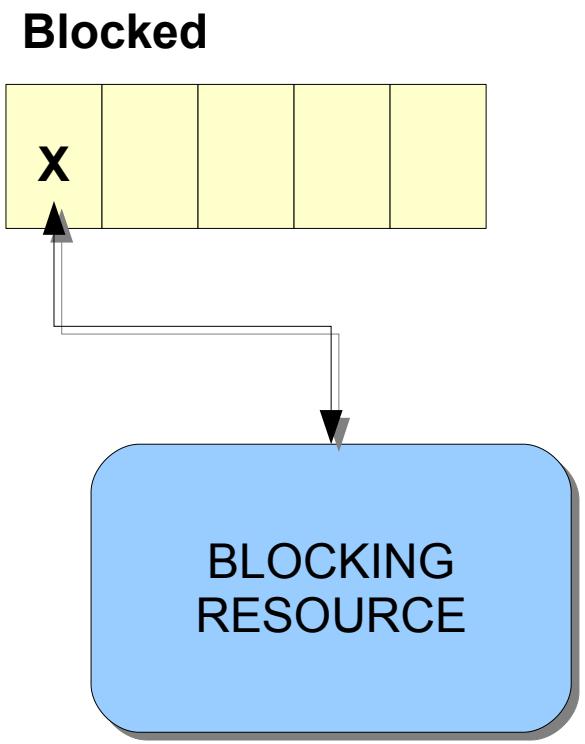
Notes from the example (II)

- Context switch might have effect:
  - Switching is not for free!:



Notes from the example (III)

- What happens if tasks are blocked?



- Task goes to blocked queue
- A pointer is set to/from blocking resource
  - MUTEX, mailbox, etc...
- Task is moved to “Ready” when the resource is free

Outline

- Basic concepts
- Real Time Operating Systems
- RTOS Design considerations
- Tasks and scheduler in RTOS
- Tasks communication and synchronization
- Example
- “Real” RTOS
- **POSIX Standard**
- Conclusions and references

POSIX Standard

- Overview
- RT Extensions

POSIX: Overview

- **P**ortable **O**perating **S**ystem **I**nterface
- Motivation
  - Portability
  - Interoperability
- Set of standards
  - Real time extensions
    - POSIX 4

POSIX.1	System Interface (basic reference standard)[a,b]
POSIX.2	Shell and Utilities[a]
POSIX.3	Methods for Testing Conformance to POSIX[a]
POSIX.4	Real-time Extensions
POSIX.4a	Threads Extensions
POSIX.4b	Additional Real-time Extensions
POSIX.6	Security Extensions
POSIX.7	System Administration
POSIX.8	Transparent File Access
POSIX.12	Protocol Independent Network Interfaces
POSIX.15	Batch Queuing Extensions
POSIX.17	Directory Services

[a] Approved IEEE standards
[b] Approved ISO/IEC standard

POSIX: RT Extensions

- Real Time Scheduling
  - 3 policies
    - FIFO
    - Round Robin
    - Other (implementable)
- Virtual Memory
  - Special functions to bound v.m. mechanisms

POSIX: RT Extensions

- Process Synchronization
  - Semaphores
    - Priority inversion possible (!)
- Shared Memory
  - can be protected by semaphores
- Signals
  - Event notification
    - signal priority
    - queued
    - data field

POSIX: RT Extensions

- Message queues
  - message priorities
  - sending and receiving can be blocking or not
- Time
  - Clock
    - resolution = nanoseconds
  - Timers
    - programmed to a certain interval
    - expiration sends a signal to the owner process

POSIX: RT Extensions

- Asynchronous I/O
  - No wait for I/O operation
  - A signal is sent when done
- Other extensions:
  - Threads
  - Timeouts
    - limit blocking periods

Outline

- Basic concepts
- Real Time Operating Systems
- RTOS Design considerations
- Tasks and scheduler in RTOS
- Tasks communication and synchronization
- Example
- “Real” RTOS
- POSIX Standard
- **Conclusions and references**

Conclusions

- RTOS : deterministic OS
- Design of RT Systems
  - Limited by:
    - Time & Resources
  - Tasks
    - TCB
  - Scheduler
    - queues
- POSIX : Portable RT systems

References

- Operating Systems, William Stallings, ed. Prentince Hall
- “Diseno de Sistemas en Tiempo Real”, Guillem Bernat, Albert Llamosi, Ramon Puijaner, UIB
- “Missconceptions about Real-Time Computing”, John A. Stankovic
- “POSIX in Real-Time”, Kevin M. Obenland
  - http://www.embedded.com/story/OEG20010312S0073
- “Real-Time POSIX: An Overview”, Michael Gonzalez Harbour, Univ. Cantabria
- OneSmartClick.Com
  - http://www.onesmartclick.com/rtos/rtos.html

The End

Thank you!