Embedded Operating Systems

Prof. Otávio Gomes
otavio.gomes@unifei.edu.br

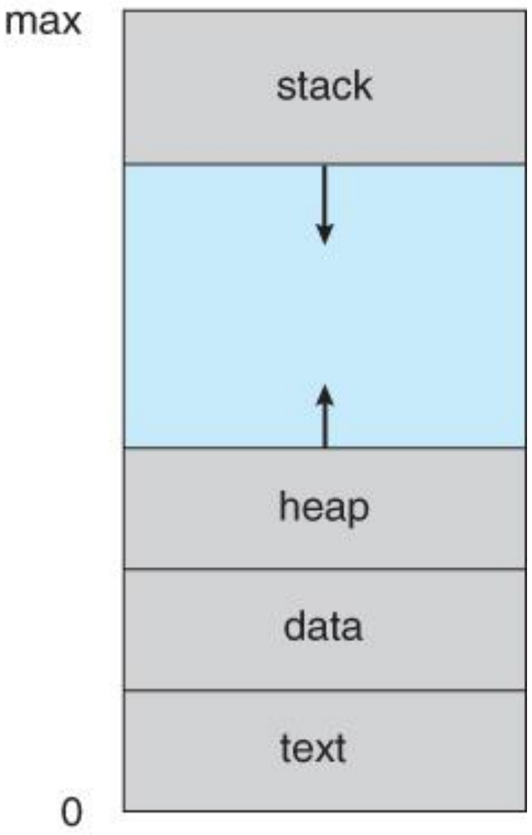
/otavio-gomes

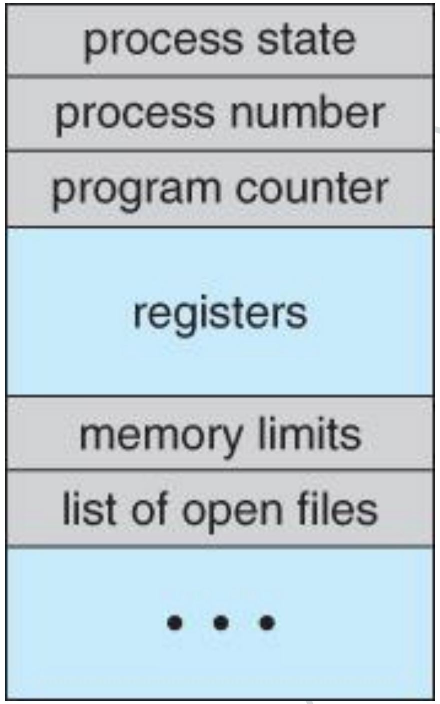
Processes

I really mean tasks.

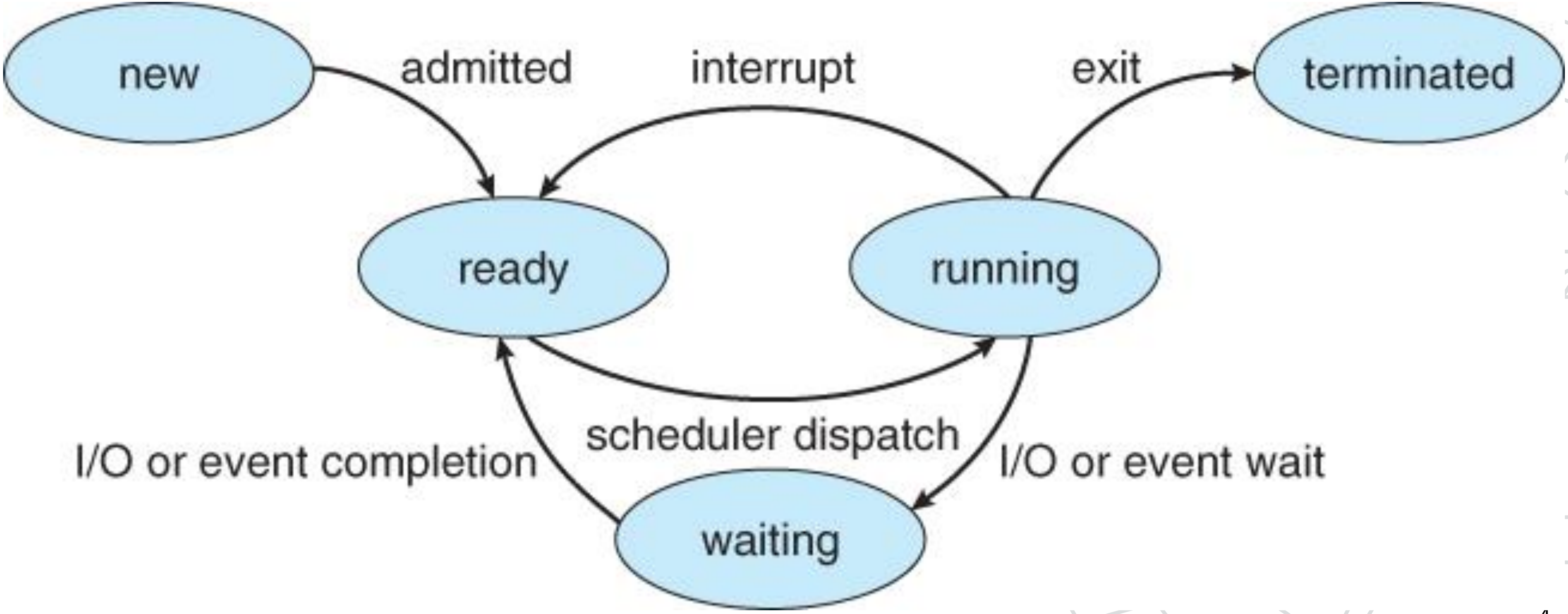
Process

- A process consists of a unit of code that can be executed, a delimited region of memory, and a set of information about its current state.

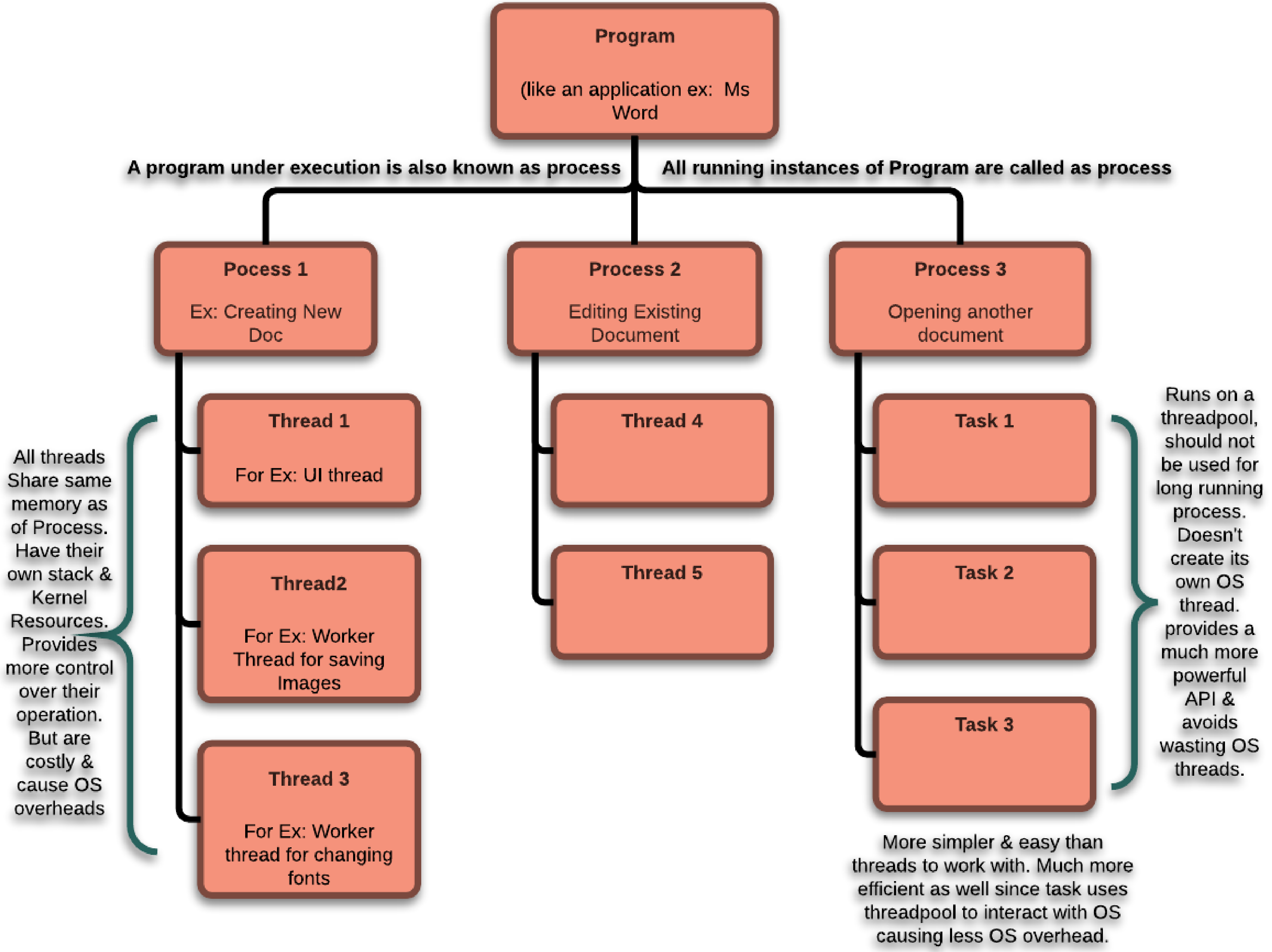




Process

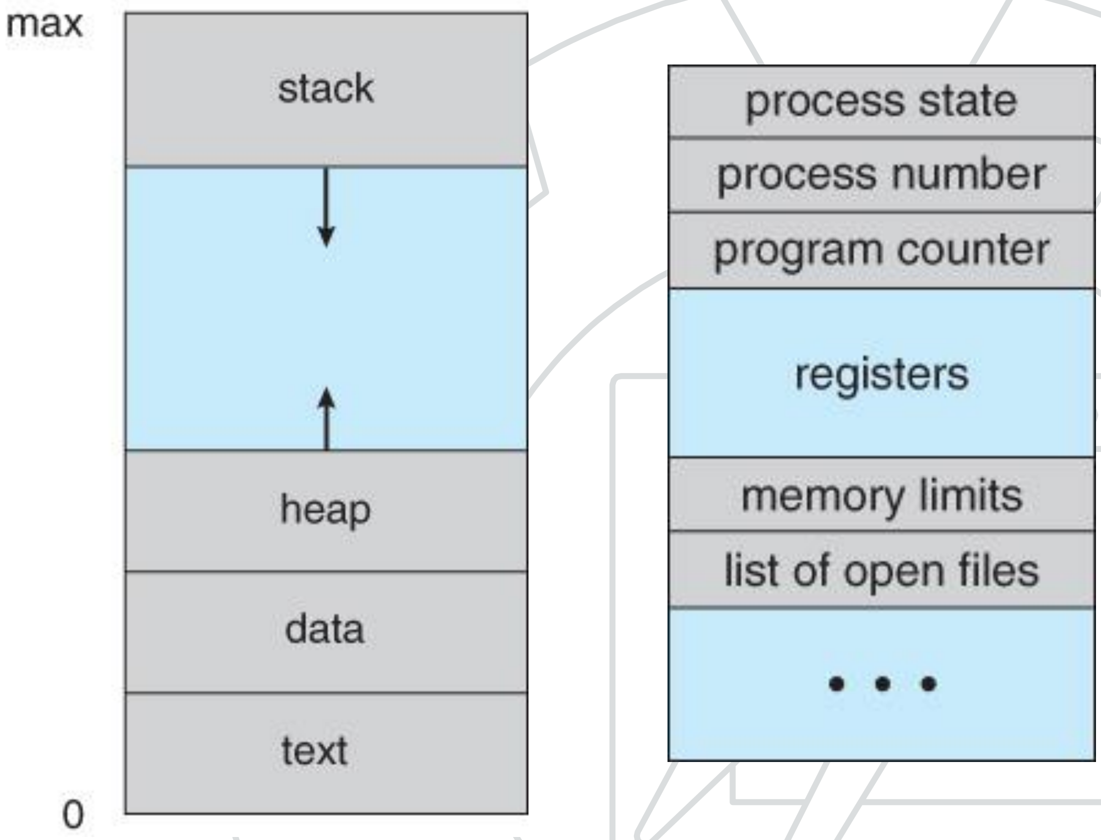

new
admitted
interrupt
exit
terminated
ready
running
scheduler dispatch
I/O or event completion
I/O or event wait
waiting

Process
vs Tasks/Threads



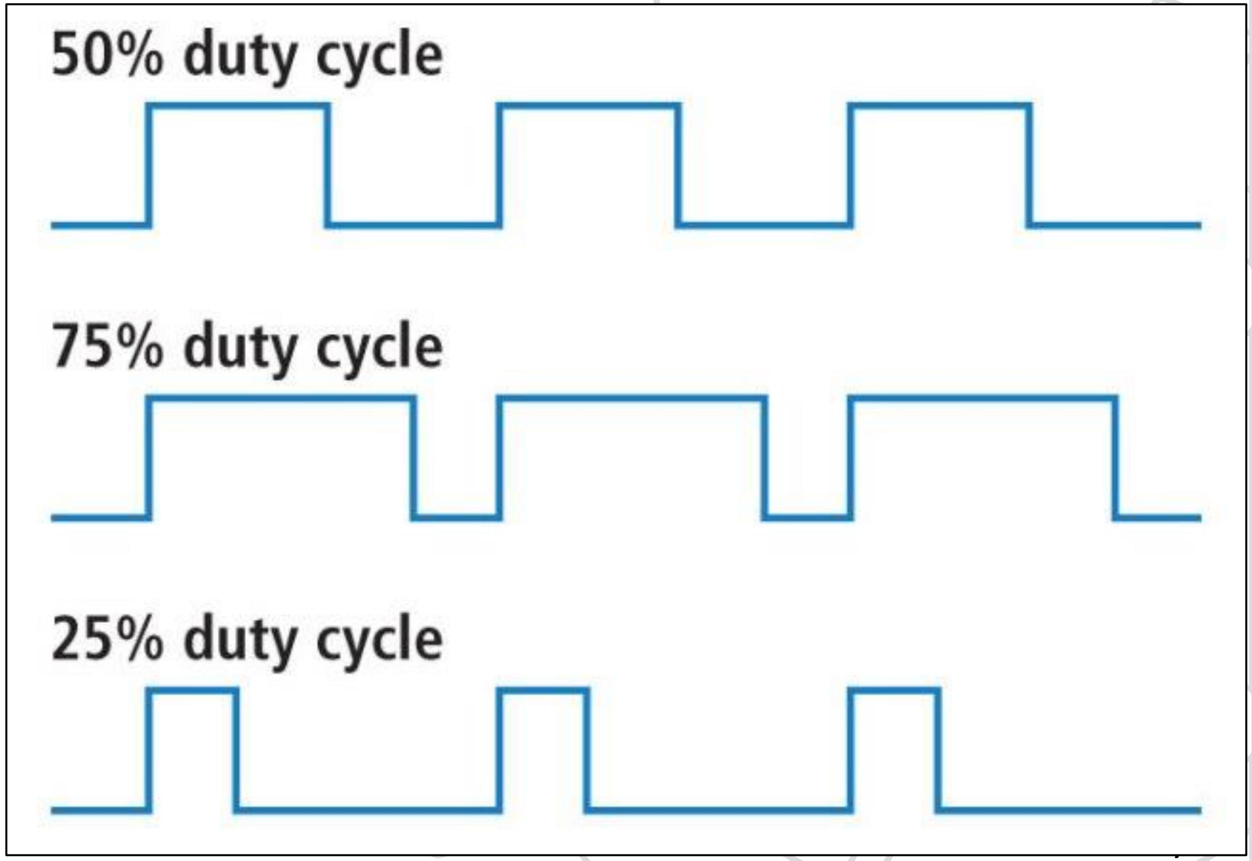
Process

- The implementation of a process is closely dependent on the **type of kernel** used and the **interfaces available** to the programmer.
- The simplest process can be represented by a function.



```c
//ponteiro para posição de I/O
#define LEDS (*((unsigned char*)0xF95))

//processo para piscar os leds
void blinkLeds (int time){
  int i;
  //liga os leds
  LEDS = 0x00;
  for(i = 0; i < time; i++){
    __asm NOP
  }
  //desliga os leds
  LEDS = 0xFF;
  for(i = 0; i < time; i++){
    __asm NOP
  }
}
```



Process

Process reschedule

Process reschedule

- How do I keep the process running?

```c
//ponteiro para posição de I/O
#define LEDS (*((unsigned char*)0xF95))

//processo para piscar os leds
void blinkLeds (int time){
  int i;
  //liga os leds
  LEDS = 0x00;
  for(i = 0; i < time; i++){
    __asm NOP
  }
  //desliga os leds
  LEDS = 0xFF;
  for(i = 0; i < time; i++){
    __asm NOP
  }
}
```

Process

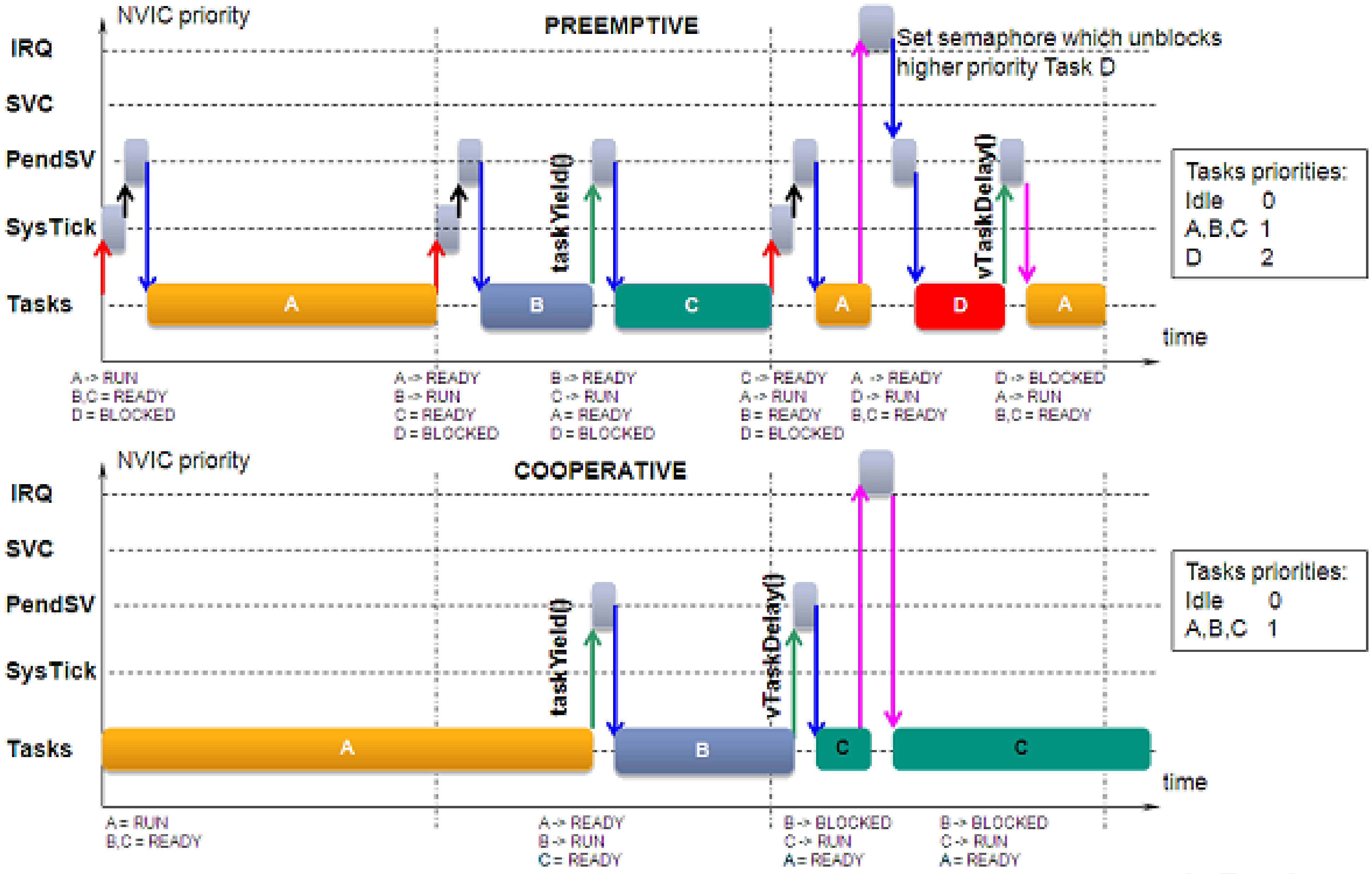
Process reschedule

- How do I keep the process running?
  - Re-run function?
  - Create a loop with n repeats?
  - Create an infinite loop?

Process reschedule

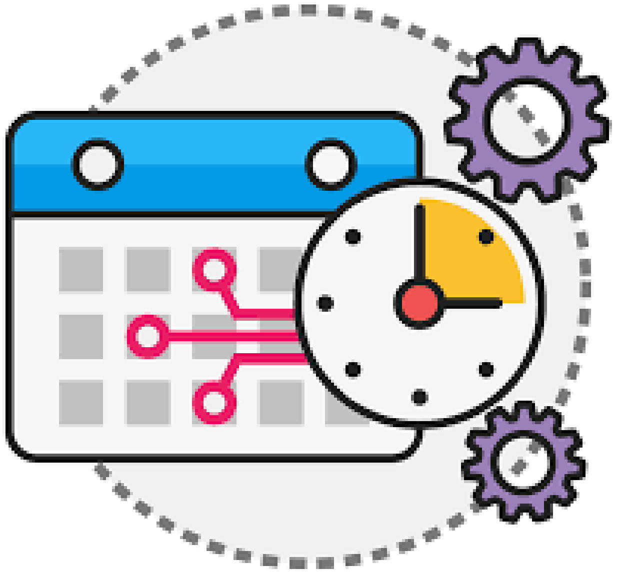
- First you must know:
  - What kind of kernel is running: preemptive or cooperative?

Process reschedule



Process reschedule

- First you must know:
  - What kind of kernel is running: preemptive or cooperative?
  - Is there a time scheduler?



Process reschedule

- Infinite Loop: Should only be used if

  1) the kernel is able to interrupt the process (preemptive);

  2) the process is the only one running on the system.

- In case 2 it does not make sense to have a kernel.

Embedded Programming: ECOP04 + ECOP14

Process – infinite loop

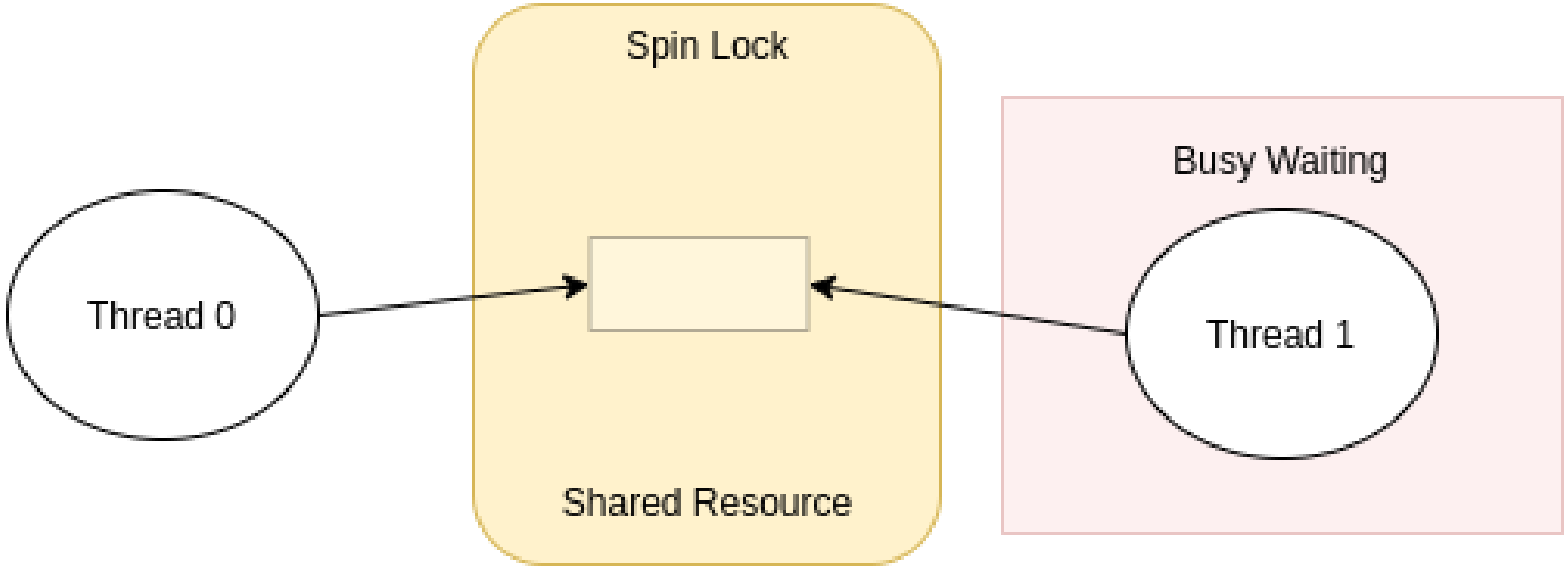
```c
//processo para piscar os leds
void blinkLeds (int time){
  int i;
  //liga os leds
  for(;;){//ever
    LEDS = 0x00;
    for(i = 0; i < time; i++){
      __asm NOP
    }
    //desliga os leds
    LEDS = 0xFF;
    for(i = 0; i < time; i++){
      __asm NOP
    }
  }
}
```

Process reschedule

- Re-run the function:
  - Allows the process to free up time for the kernel to perform other functions.
  - Must be used in <u>cooperative kernel</u>.
  - Develop small tasks.
- When possible, one should <u>not use delay routines</u>, leaving time counting for the kernel.
- In this way the system can perform more useful tasks while waiting

Process reschedule

- When possible, one should <u>not use delay routines</u>, leaving time counting for the kernel.
- In this way the system can perform more useful tasks while waiting



Process re-execute

```c
//Original
//processo para piscar os leds
void toggleLeds (int time){
  int i;
  LEDS = 0x00;
  for(i = 0; i < time; i++){
    __asm NOP
  }
  //desliga os leds
  LEDS = 0xFF;
  for(i = 0; i < time; i++){
    __asm NOP
  }
}
```

- Wasting time;
- Sync.

- Wasting time;
- Sync.

Process re-execute

```c
//Omissão das rotinas de tempo 1
//processo para piscar os leds
void toggleLeds (int time){
   int i;
   //liga os leds
   LEDS = 0x00;
   //desliga os leds
   LEDS = 0xFF;
}

//Não funciona, deve ligar em uma chamada e desligar em outra
```

Sync?

Process re-execute

```c
//Omissão das rotinas de tempo 2
//processo para piscar os leds
void toggleLeds (int time){
    int i;
    LEDS = ~LEDS;
}

//Não funciona bem, os tempos não são respeitados
```

Process re-execute

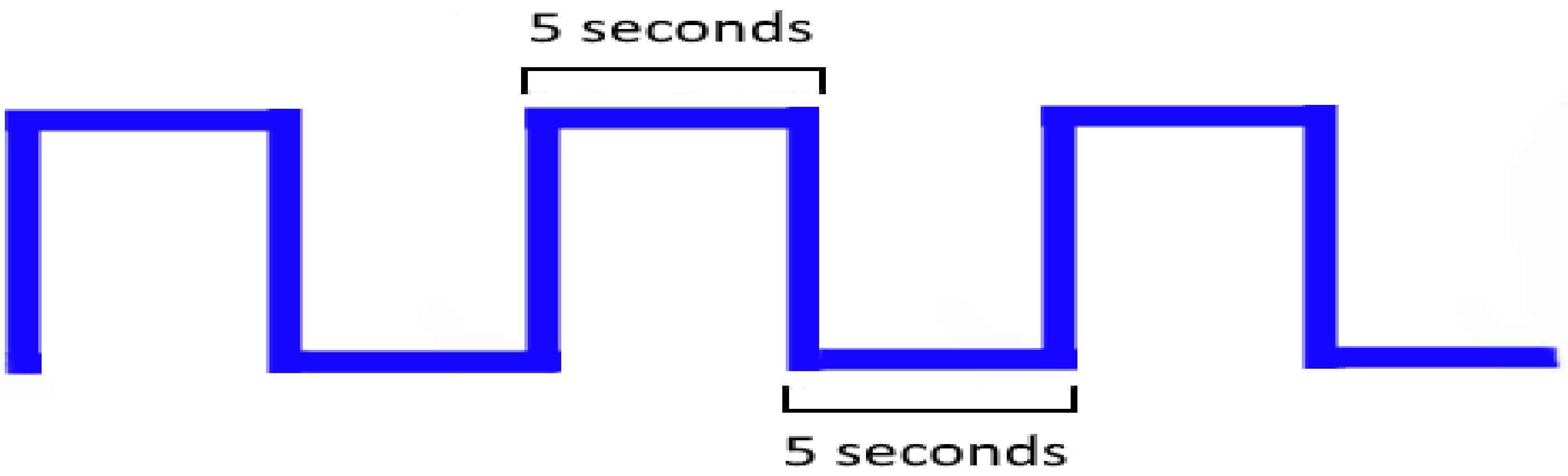
```c
//Omissão das rotinas de tempo 3
//processo para piscar os leds
void toggleLeds (int time){
  int i;
  static int lastTime;
  if ( (now() - lastTime) >= time){
    LEDS = ~LEDS;
    lastTime = now();
  }
}

// a função now() deve retornar o horário em unidades de segundo/milisegundo
// static não perde o valor entre as chamadas
// ECOP14 – Debounce de teclas
```

```c
//Omissão das rotinas de tempo 3
//processo para piscar os leds
void toggleLeds (int time){
  int i;
  static int lastTime;
  if ( (now() - lastTime) >= time){
    LEDS = ~LEDS;
    lastTime = now();
  }
}
```

Process re-execute

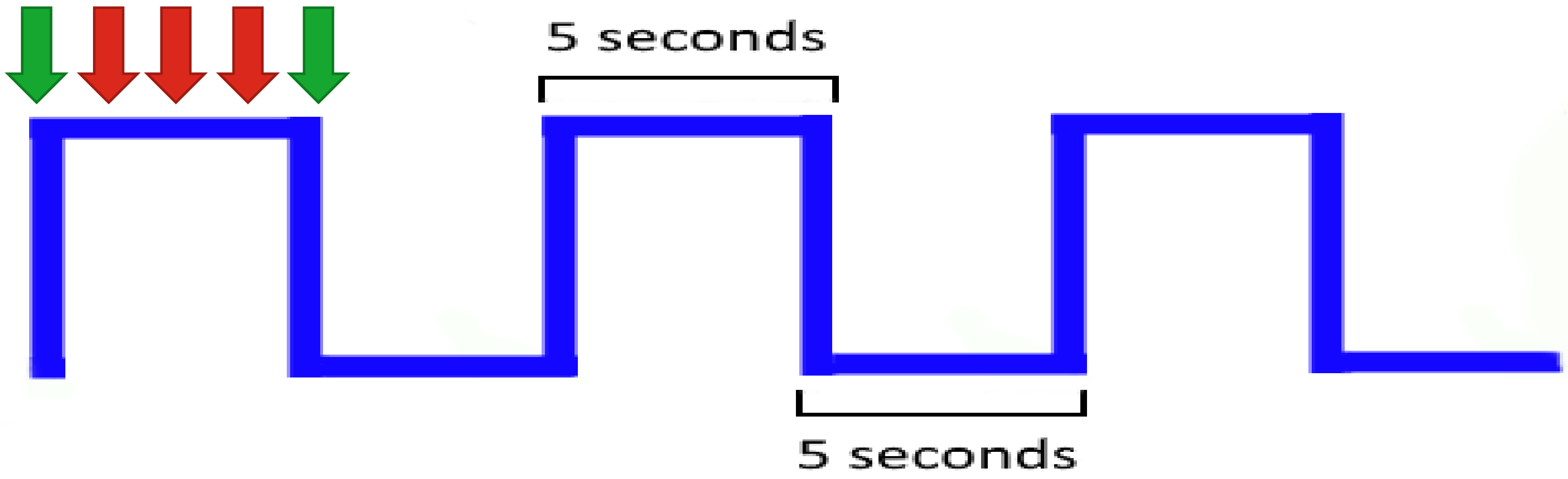
time = 5s

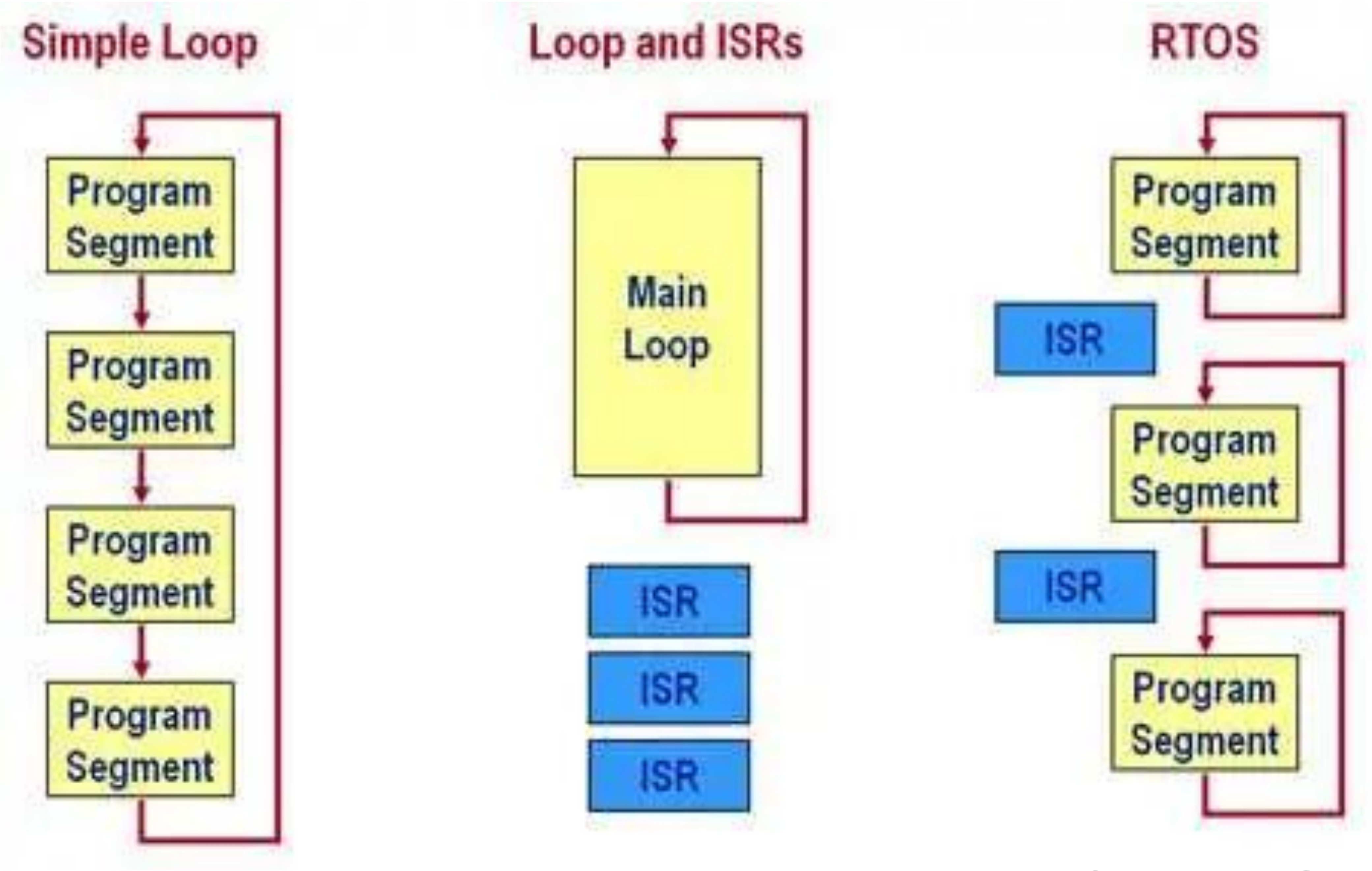


```c
//Omissão das rotinas de tempo 3
//processo para piscar os leds
void toggleLeds (int time){
  int i;
  static int lastTime;
  if ( (now() - lastTime) >= time){
    LEDS = ~LEDS;
    lastTime = now();
  }
}
```

Process re-execute

time = 5s





ISR: Interrupt Service Routine

Process definition

In C language and for this class

Process

- As mentioned the process is, in principle, a function that must be executed when the process is called.
- In addition there are several important information that must be aggregated to be able to manage the process
  - Priority, runtime, name, reserved memory region, etc.
- In general **a structure** is used to aggregate all this information.

```c
typedef int (*ptrFunc)(void);

//our new process
typedef struct {
   char* nomeDoProcesso;
   ptrFunc funcao;
   int prioridade;
   int tempo;
}process;
```

Process Control Block (PCB)

Process

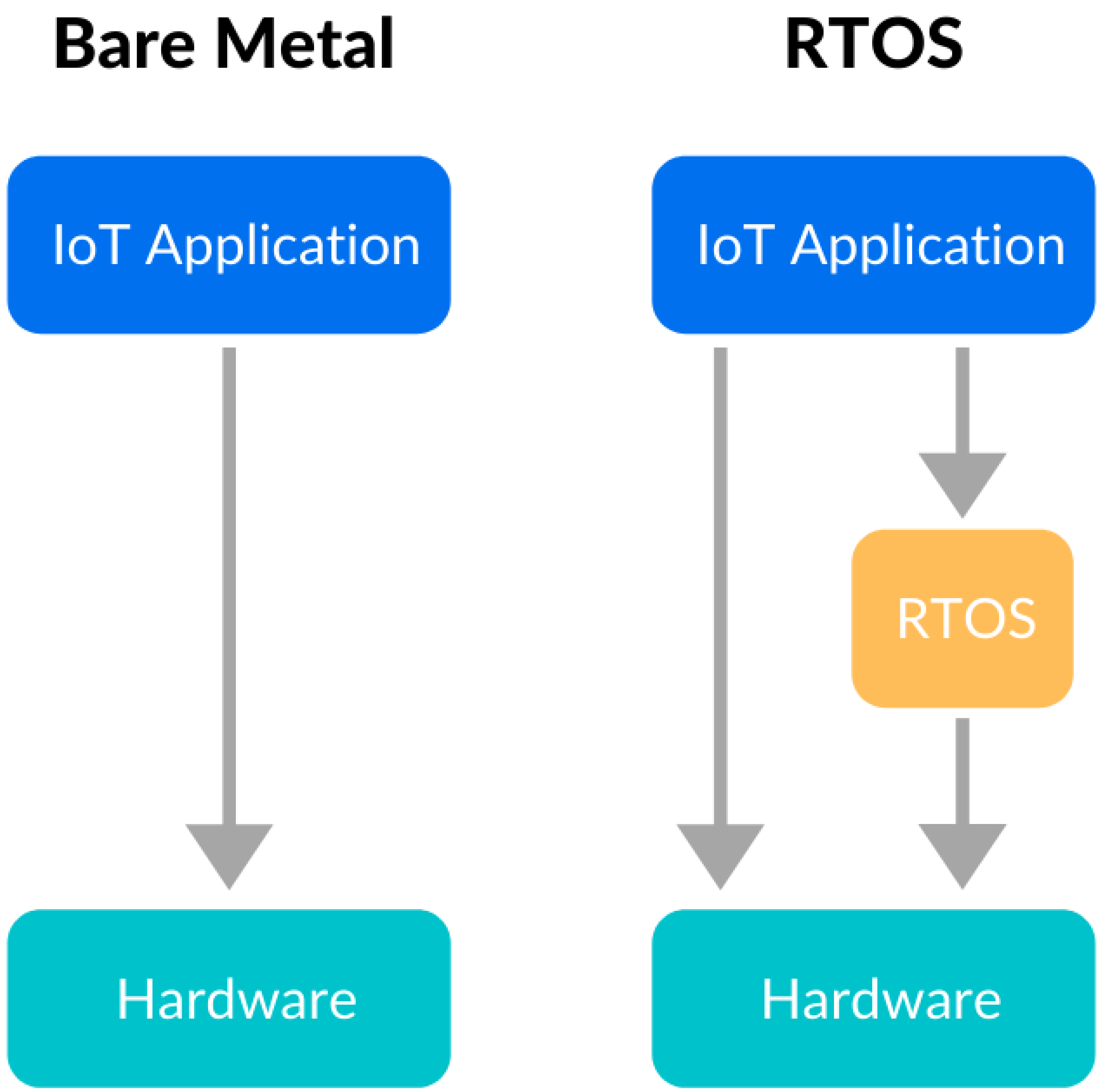
The kernel

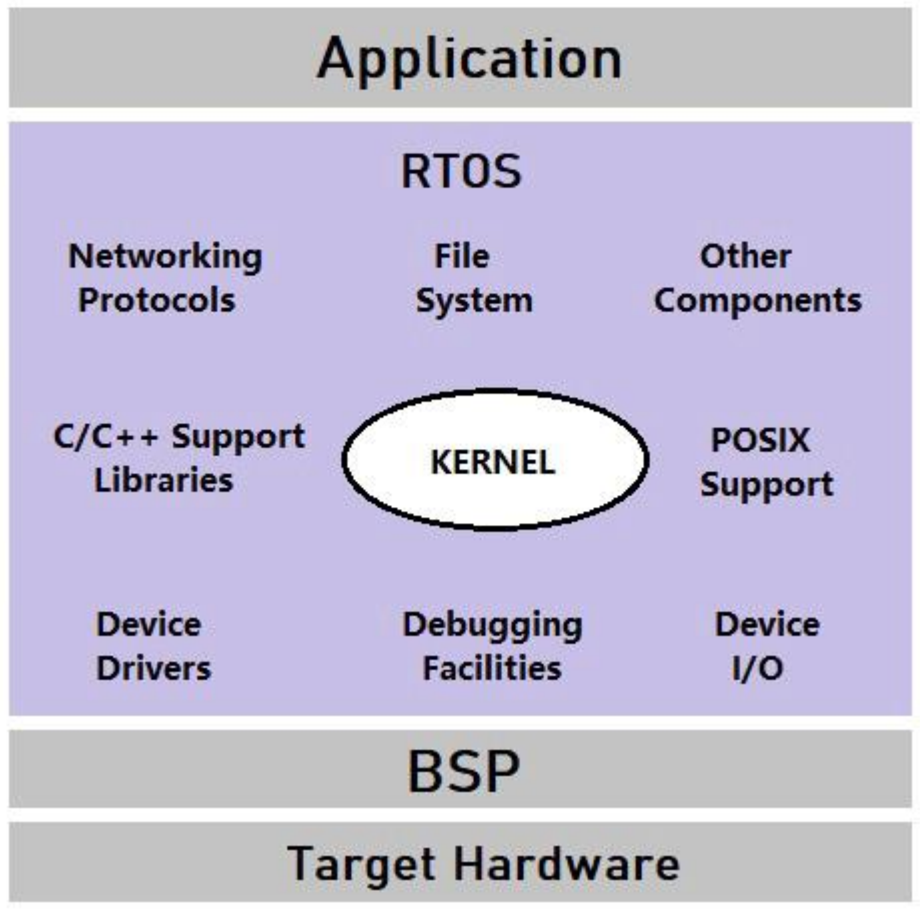
Process management core

Kernel

- In the computer field, the kernel is the part of the code responsible for manage the interface between the hardware and the application.
- The most critical hardware features in this regard are processor, memory and drivers.

Kernel responsibilities



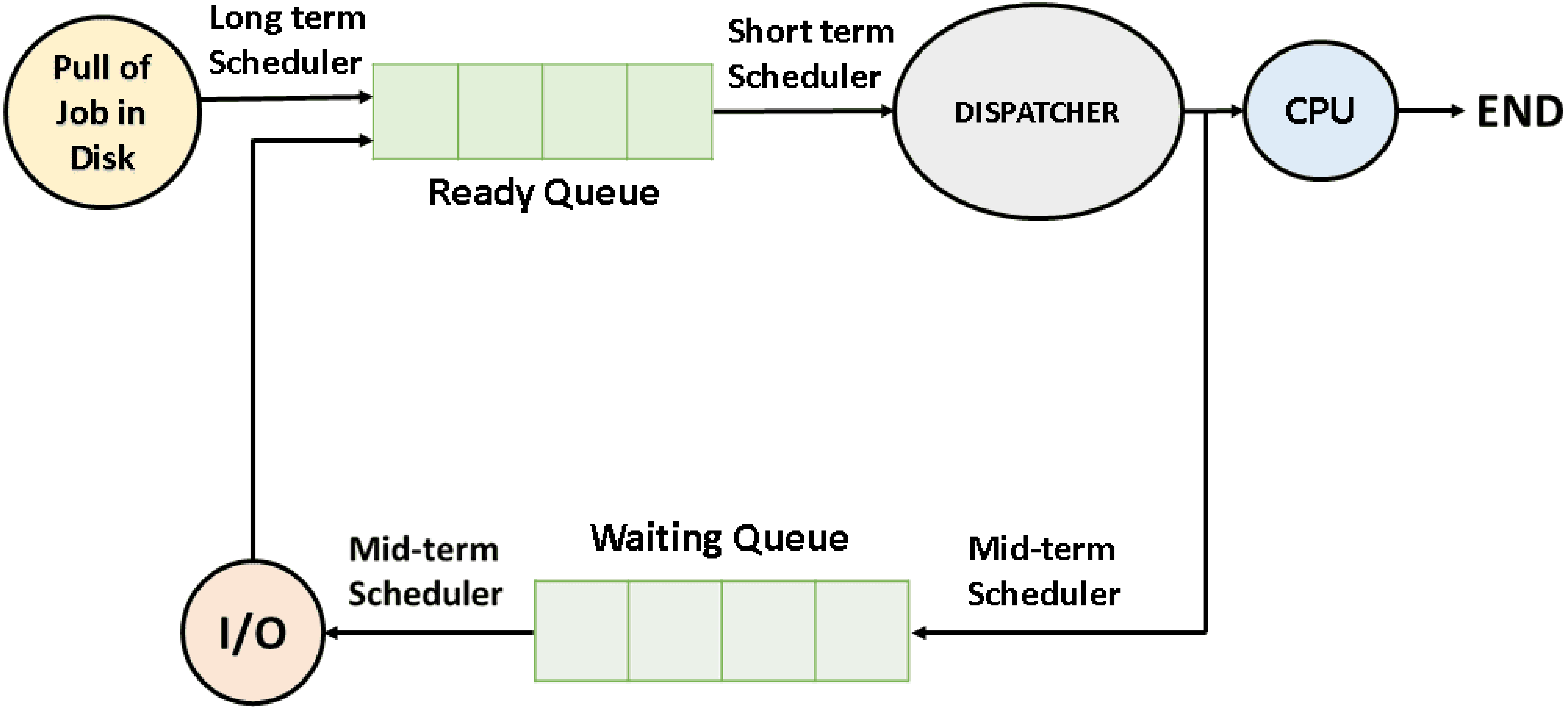


Processes management

The management can be done in lots of different ways

- We'll be using a **circular buffer** (process pool).
- The access to this buffer must be restricted to the kernel.

Processes management



Created by NotesJam

The kernel

First implementation

The kernel

- In this first example we will build the main part of our kernel.
- It should have a way to store which functions are needed to be executed and in which order.
- This will be done by a static vector of pointers to function
  - We're not using the structs, only the function pointer

```
//pointer function declaration
typedef void(*ptrFunc)(void);
//process pool
static ptrFunc pool[4];
```

The kernel

- Each process is a function with the same signature of **`ptrFunc`**

```
void tst1(void){
    printf("Process 1\n");
}
void tst2(void){
    printf("Process 2\n");
}
void tst3(void){
    printf("Process 3\n");
}
```

The kernel

- The kernel itself consists of three functions:
  - One to initialize all the internal variables
  - One to add a new process
  - One to execute the main kernel loop

```c
//kernel internal variables
ptrFunc pool[4];
int end;

//kernel function's prototypes
void kernelInit(void);
void kernelAddProc(ptrFunc newFunc);
void kernelLoop(void);
```

The kernel

```c
//kernel function's implementation
void kernelInit(void){
   end = 0; //simplified - It is not a circular buffer
}

void kernelAddProc(ptrFunc newFunc){
   if (end <4){
      pool[end] = newFunc;
      end++;
   }
}
```

The kernel

```c
//kernel function's implementation
void kernelLoop(void){
    int i;
    for(;;){
        //cycle through the processes
        for(i=0; i<end; i++){
            (*pool[i])();
        }
    }
}
```

The kernel

```c
//main loop
void main(void){

  kernelInit();

  kernelAddProc(tst1);
  kernelAddProc(tst2);
  kernelAddProc(tst3);

  kernelLoop();
}
```

Scheduler

AKA: the real boss.

Scheduler

- The scheduler is responsible for choosing which is the next process to be executed.
- There are some parameters to consider:
  - **Throughput**: number of processes per time.
  - Latency:
    - **Turnaround time** - time between the beginning and end of a process.
    - **Response time**: value between a request and the first response of the process.
    - **Fairness / Waiting Time** - allow an equal amount of time for each process.

Scheduler

- First in first out
- Shortest remaining time
- Fixed priority pre-emptive scheduling
- Round-robin scheduling
- Multilevel queue scheduling

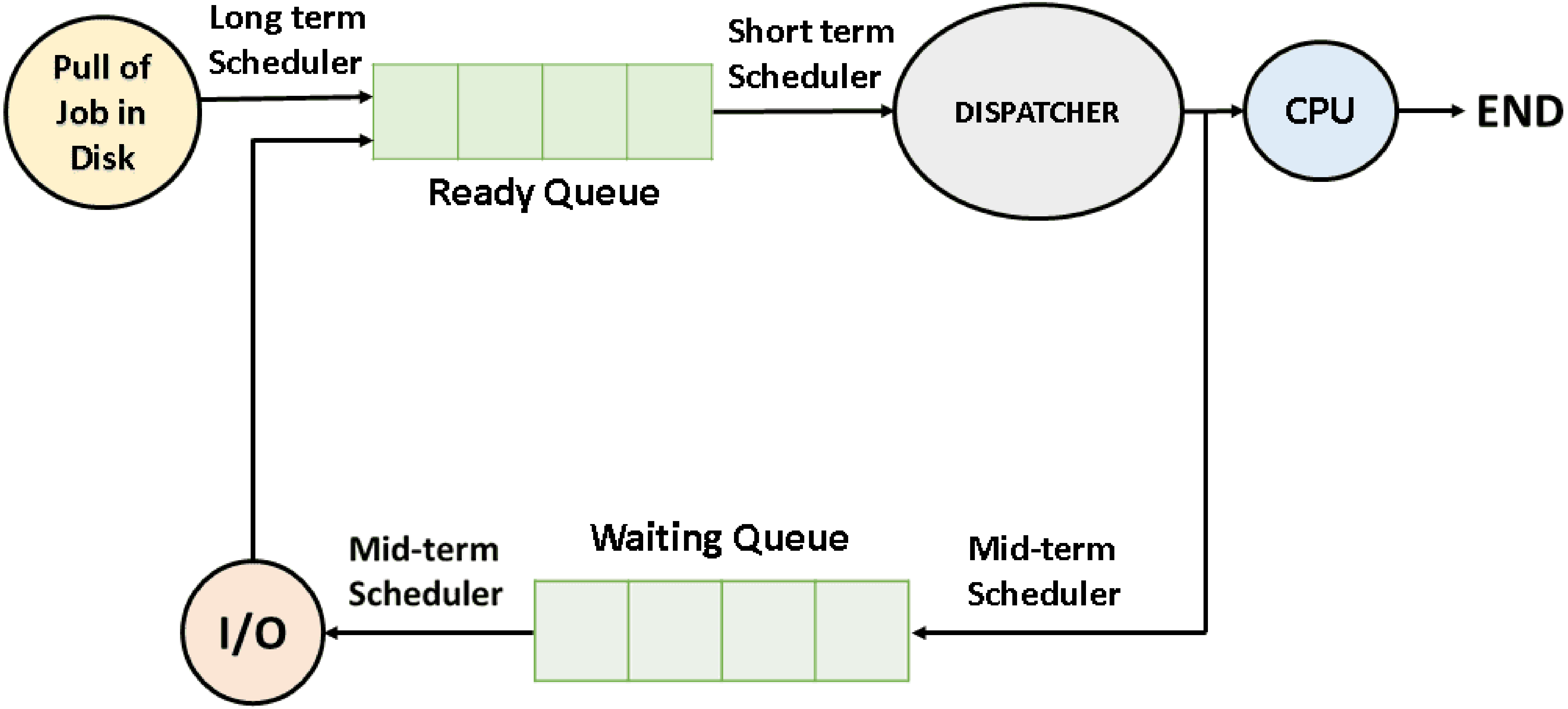
Scheduler

Scheduling algorithm	CPU Overhead	Throughput	Turnaround time	Response time
First In First Out	Low	Low	High	Low
Shortest Job First	Medium	High	Medium	Medium
Priority based scheduling	Medium	Low	High	High
Round-robin scheduling	High	Medium	Medium	High
Multilevel Queue scheduling	High	High	Medium	Medium

Scheduler

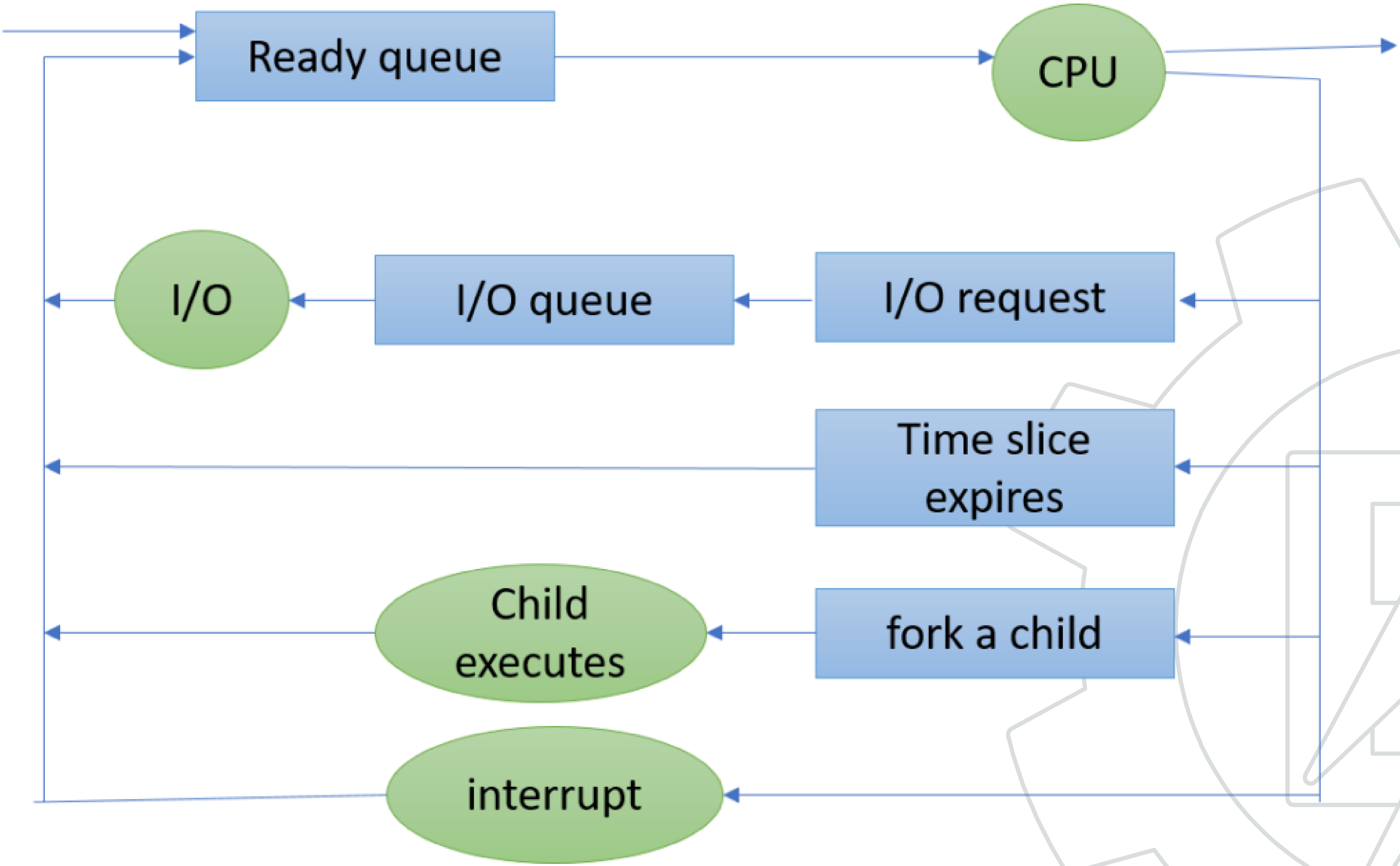
Operating System	Preemption	Algorithm
Amiga OS	Yes	Prioritized Round-robin scheduling
FreeBSD	Yes	Multilevel feedback queue
Linux pre-2.6	Yes	Multilevel feedback queue
Linux 2.6-2.6.23	Yes	O(1) scheduler
Linux post-2.6.23	Yes	Completely Fair Scheduler
Mac OS pre-9	None	Cooperative Scheduler
Mac OS 9	Some	Preemptive for MP tasks, Cooperative Scheduler for processes and threads
Mac OS X	Yes	Multilevel feedback queue
NetBSD	Yes	Multilevel feedback queue
Solaris	Yes	Multilevel feedback queue
Windows 3.1x	None	Cooperative Scheduler
Windows 95, 98, Me	Half	Preemptive for 32-bit processes, Cooperative Scheduler for 16-bit processes
Windows NT (XP, Vista, 7, 2k)	Yes	Multilevel feedback queue

Scheduler and Dispatcher



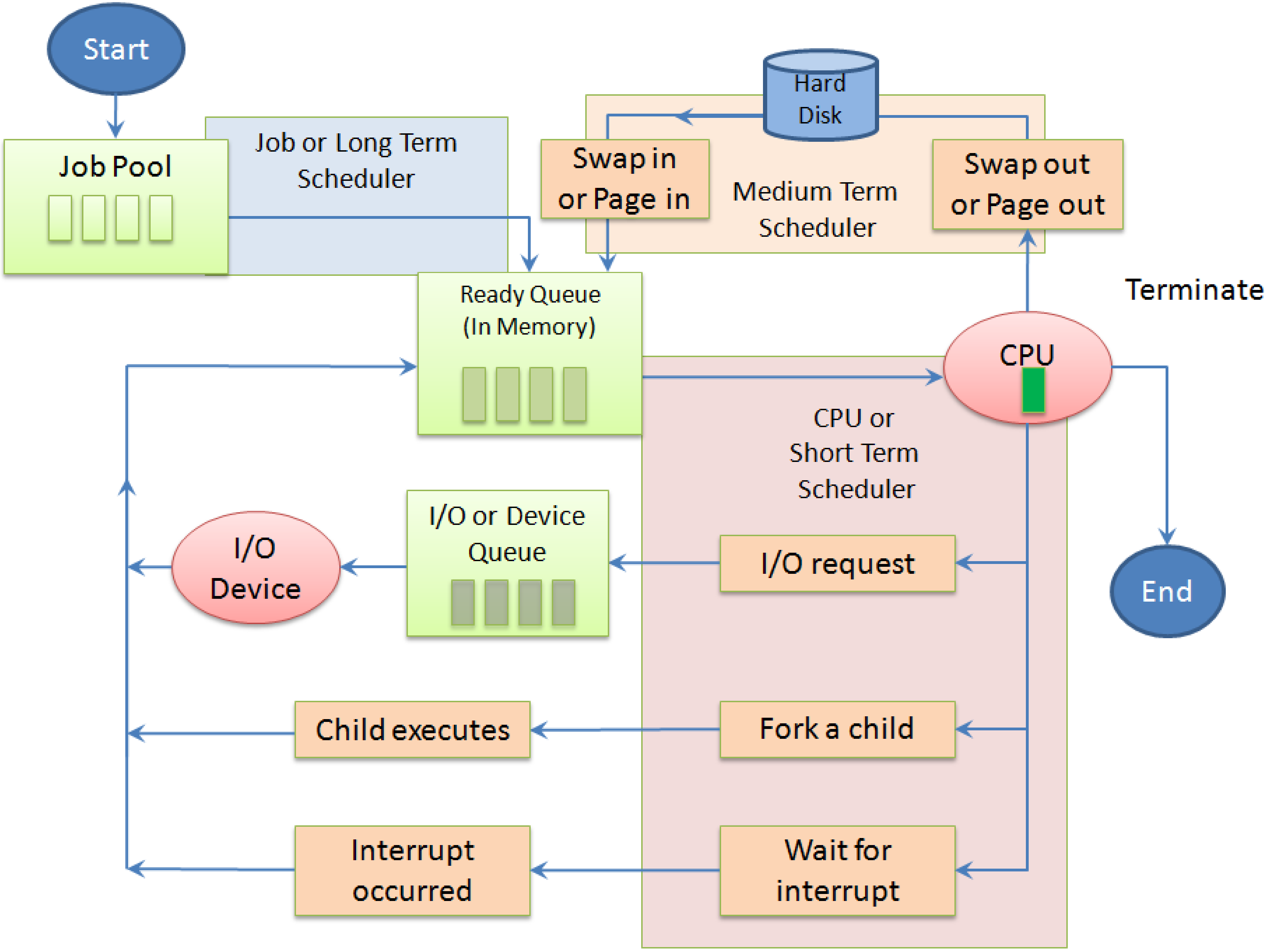
Created by NotesJam

Scheduler


Ready queue
CPU
I/O
I/O queue
I/O request
Time slice expires
Child executes
fork a child
interrupt

Scheduler

General Purpose Device

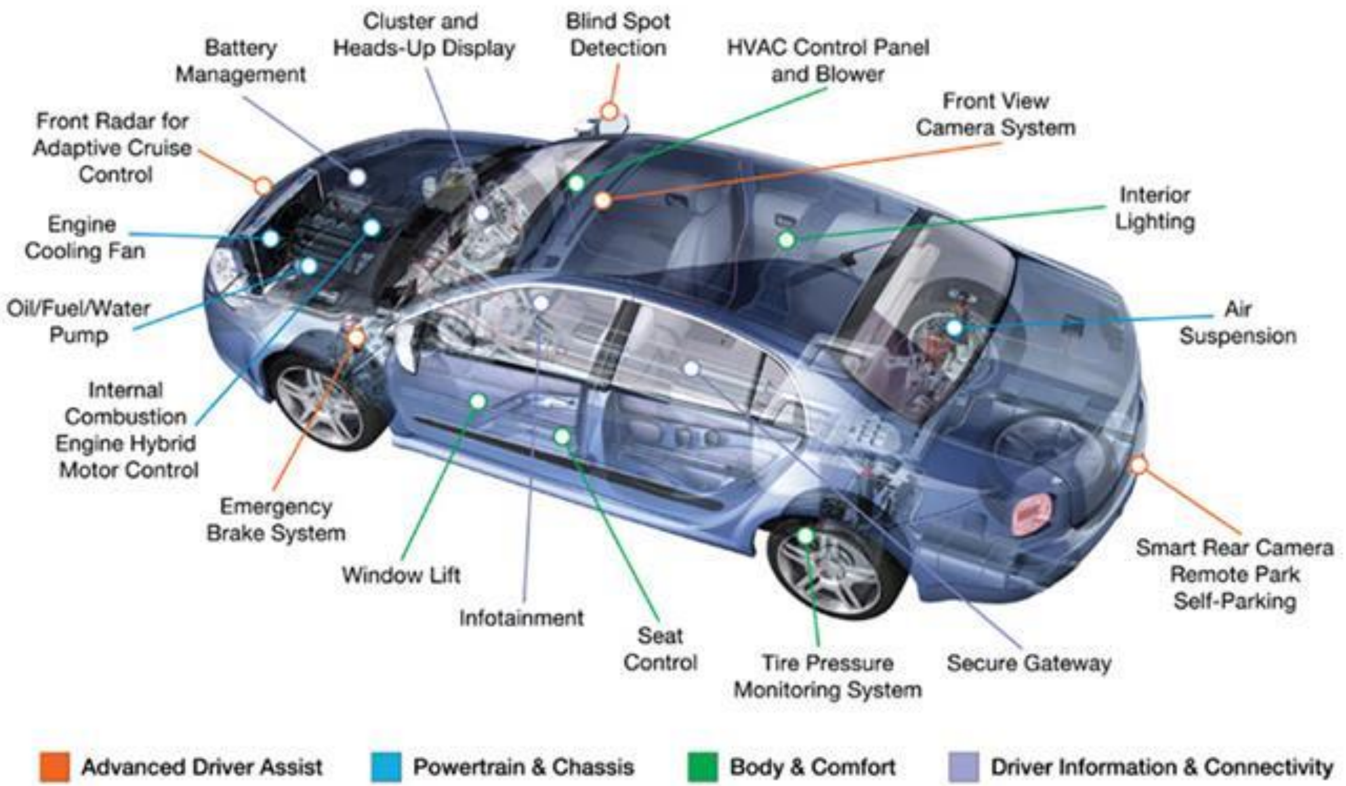


Scheduler

- Considerations for the embedded environment
  - With computational low resources, a very complex algorithm can undermine the processing capacity very quickly. <u>**Simpler algorithms are preferred**</u>.
  - Real-time systems have some needs that are usually only met by **"unfair" schedulers** that can privilege some processes.
    - Ex: priority based scheduler

Scheduler

- Why use priority based schedulers?
  - Simple and deterministic
- How can I choose a process priority?
  - Function importance
  - Shared resources
  - Mathematical model (RMS)



Rate-Monotonic Scheduling (RMS)

- Static priority <u>for each process</u>
- Priorities are given according to the <u>process cycle</u>
  - Process with **bigger** cycle has a **lower** priority
- Given that there are no resource shared between processes:

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq n\left(2^{\frac{1}{n}} - 1\right)$$

  - When n tends to infinite U must be less than 69,3%

Kernel impact on schedulers

- **Pre-emption**
  - It allows the kernel to pause a process to run a second without changing the code's variables and flow (**time slice** - *quantum*).
  - Requires hardware support due to **interruptions**
  - Programmed in assembly
- **Cooperative**
  - It is necessary that the processes **end up** giving opportunity for other processes to be executed by the processor
  - Infinite loops can lock the entire system
  - Can be programmed entire in C and requires no special hardware

Scheduler

- Rescheduling of processes
  - For a **cooperative kernel** it is important that all processes terminate voluntarily to release CPU space to the other processes.
  - In cases where the process needs to run constantly it should be rescheduled in the CPU

Scheduler Characteristics

- A typical <u>pre-condition for hard real-time</u> periodic processes is that they should <u>**always meet their deadlines**</u>.
- A static approach calculates (or <u>**pre-determines**</u>) schedules for the system.
- It requires <u>prior knowledge</u> of a process's characteristics but requires little run-time overhead.

Scheduler Characteristics

- Dynamic method determines schedules **at run-time**
- It has higher run-time cost but can give greater processor utilization.
  - Whether dynamic algorithms are appropriate for hard real-time systems is, however, a matter of some debate.

Scheduler Characteristics

- Certainly in safety critical systems it is reasonable to argue that no event should be unpredicted and that schedulability should be guaranteed before execution.
  - This implies the use of a static scheduling algorithm

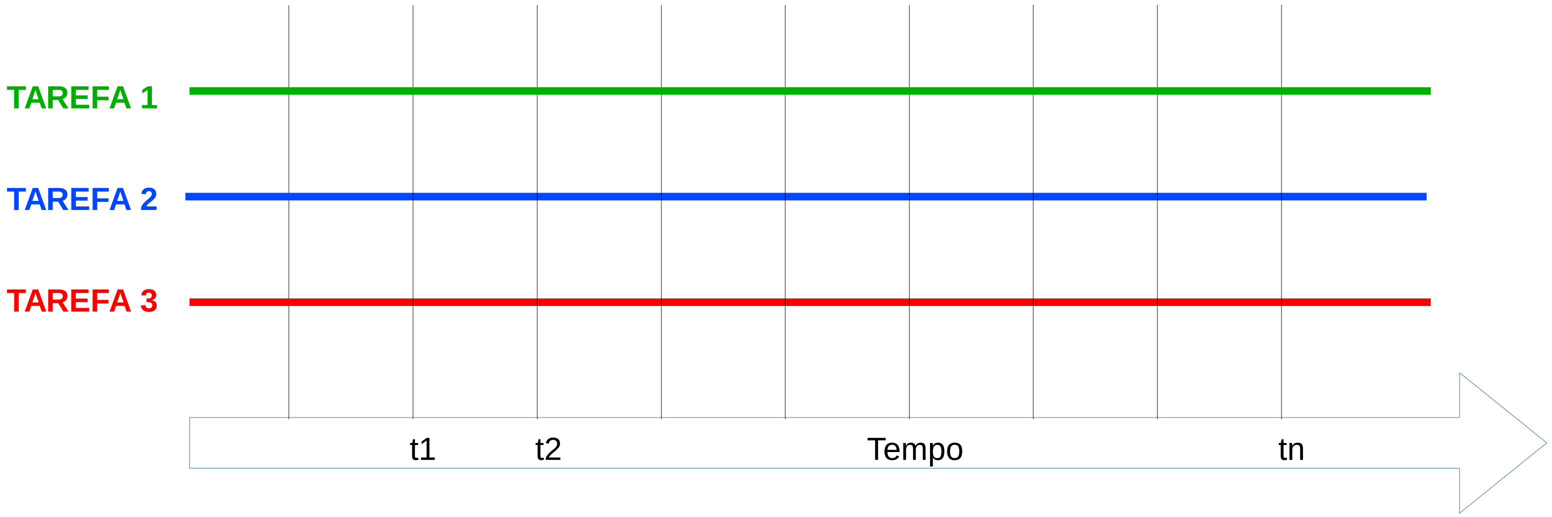
Scheduler Characteristics

- Dynamic approaches uses:
  - they are particularly appropriate to soft systems;
  - they could form part of an **error recovery procedure** for missed hard deadlines;
  - they may have to be used if the application's requirements fail to provide a worst case upper limit

Preemption

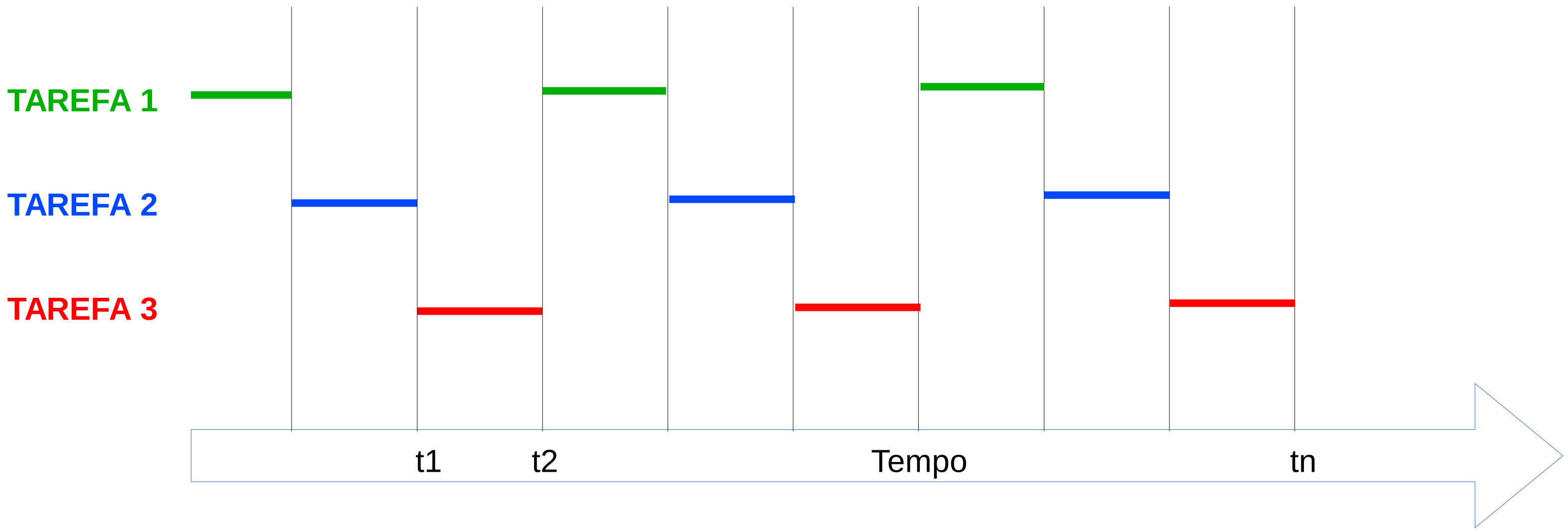
Multitask

- In a multitasking system, we get the <u>impression</u> that all tasks are running at the same time.



Multitask

- But because the processor can only execute one task at a time (considering a CPU with only one core), a switch between tasks is performed:



Colaborative:

- I/O request
- Finish actions, batch
- Infinite Loop is a problem

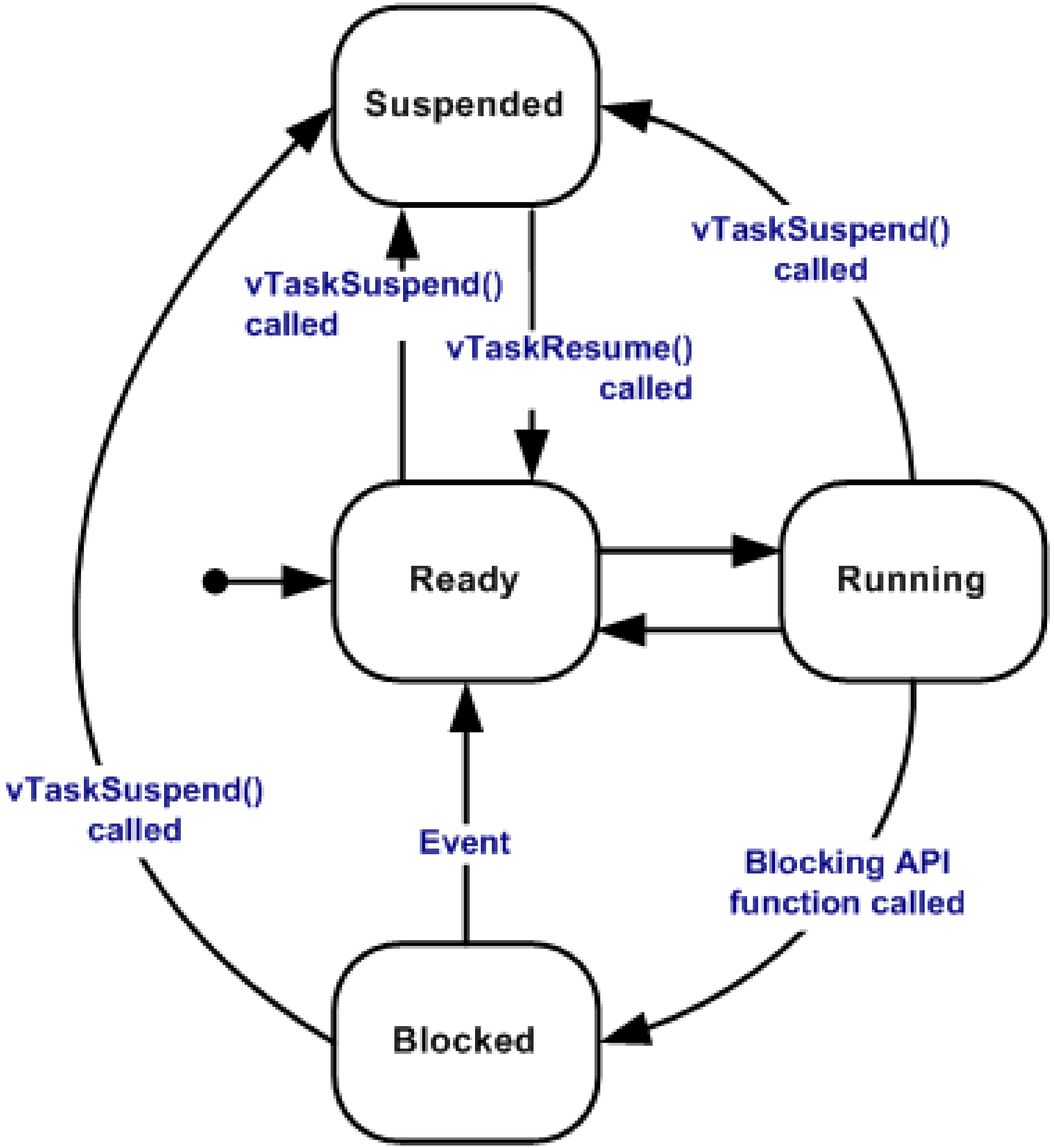
Preemptive:

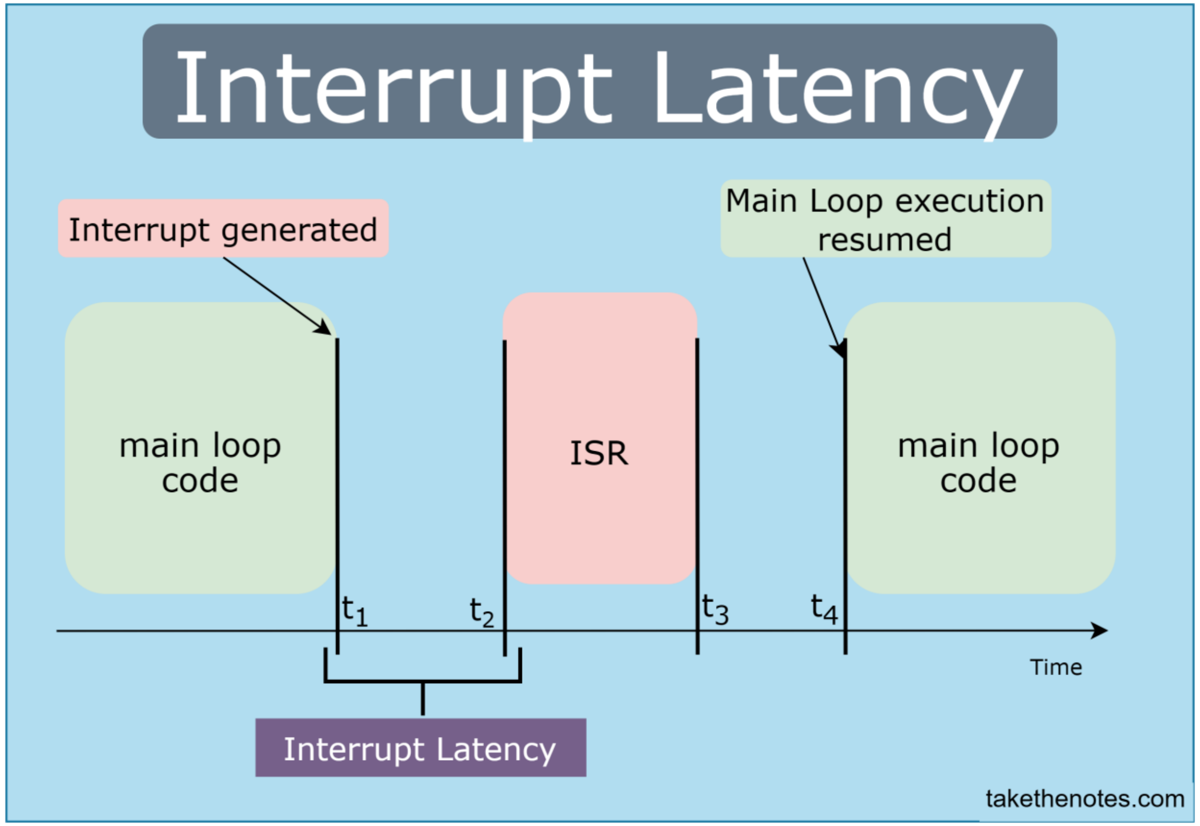
- Pause process execution
- Time slice
- Interrupt Service Routine (ISR)

Multitask

- This switch or task change can happen in different situations:
  - A task may block waiting for a resource (eg serial port) to be available or an event to occur (eg receive a packet from the USB interface).
  - A task can sleep (block) for a while.
  - A task can be unintentionally suspended by the kernel. In this case, we call the kernel preemptive.
- This change of tasks is also called context switching.

Multitask


Suspended
vTaskSuspend() called
vTaskSuspend() called
vTaskResume() called
Ready
Running
vTaskSuspend() called
Event
Blocking API function called
Blocked


Interrupt Latency
Interrupt generated
Main Loop execution resumed
main loop code
ISR
main loop code
t_1
t_2
t_3
t_4
Time
Interrupt Latency
takethenotes.com

Context switch

- While a task is running, it has a certain context (stack, CPU registers, etc).
- By changing the running task, the kernel saves the context of the task to be suspended and retrieves the context of the next task to be executed.
- The control of the context of each of the tasks is carried out through a structure called TCB (Task Control Block).

Multitask

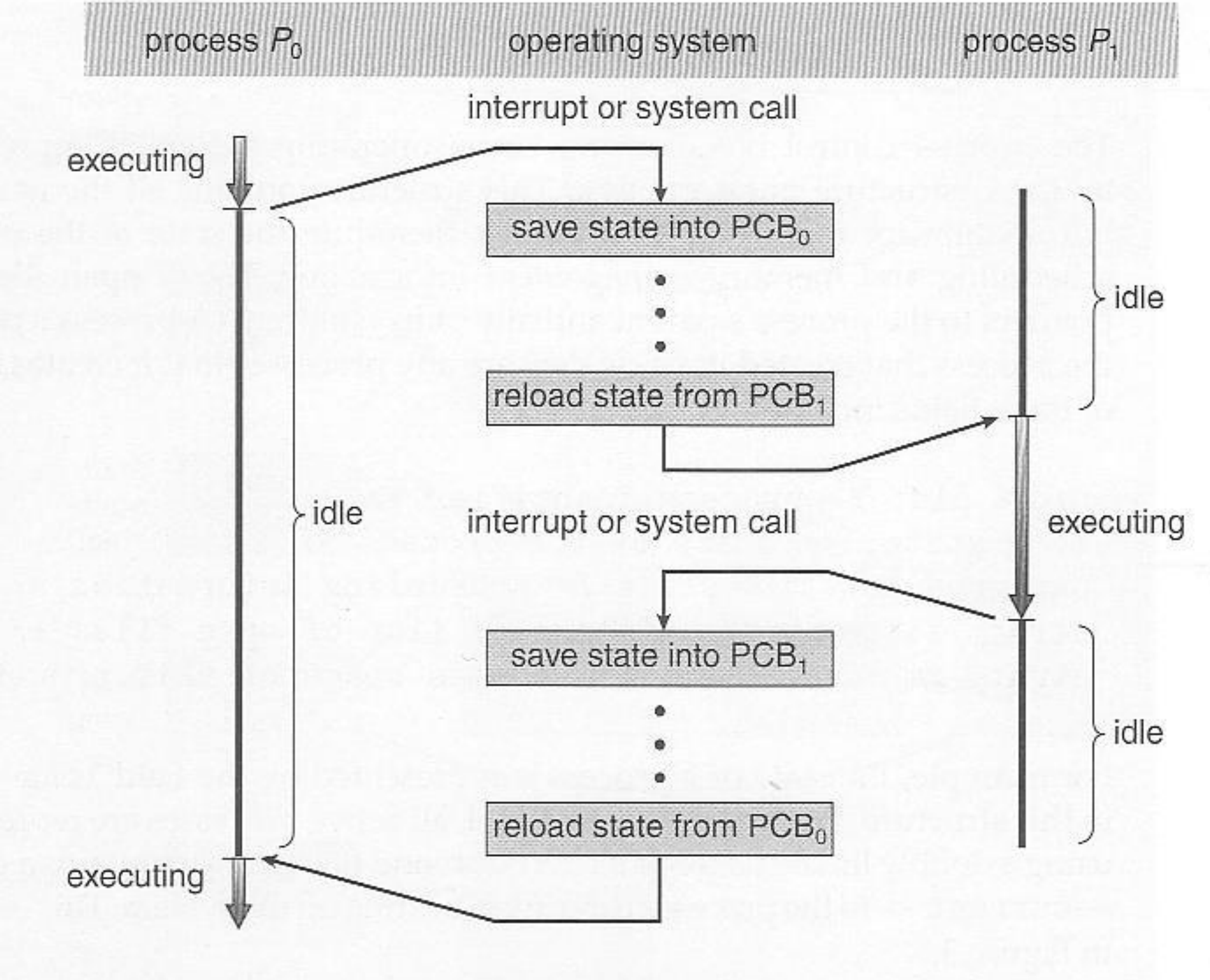
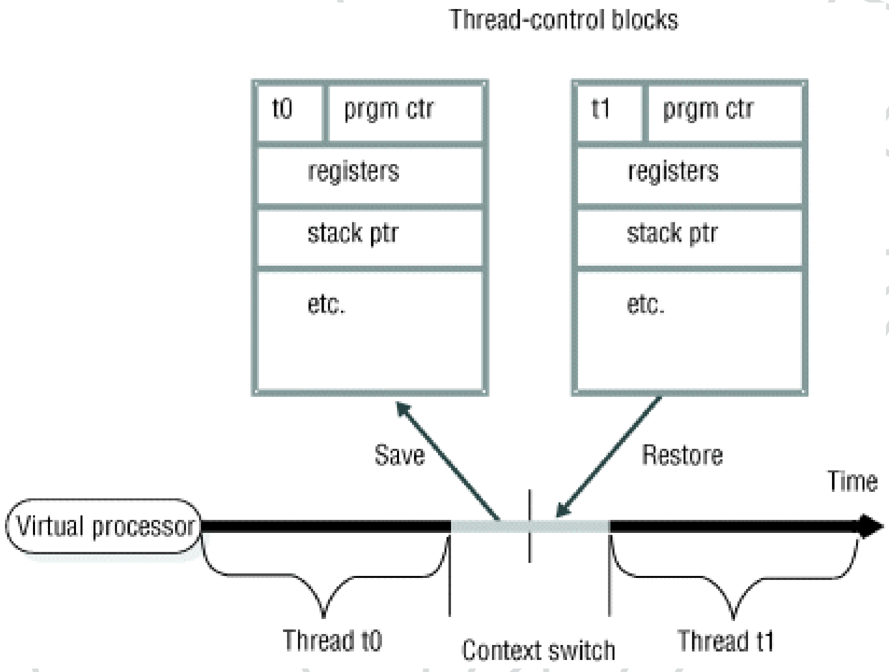


Figure 3.4 Diagram showing CPU switch from process to process.



Context switch

		Interrupção	Salvando contexto	Mudança no SP	Restauração do contexto
Registros da CPU	PC	0x32	-	-	0x3a
	Acc	0x02	-	-	0x12
	CCR	0xd4	-	-	0x00
	SP	0xad	0xaa	0xa2	0xa5
Memória utilizada como pilha	0xa0				
	0xa1				
	0xa2				
	0xa3	0x00	0x00	0x00	
	0xa4	0x12	0x12	0x12	
	0xa5	0x3a	0x3a	0x3a	
	0xa6	var3	var3	var3	var3
	0xa7	var4	var4	var4	var4
	0xa8				
	0xa9				
	0xaa				
	0xab		0xd4	0xd4	0xd4
	0xac		0x02	0x02	0x02
	0xad		0x32	0x32	0x32
	0xae	var2	var2	var2	var2
	0xaf	var1	var1	var1	var1

Ponteiro de pilha

Dados do processo A

Dados do processo B

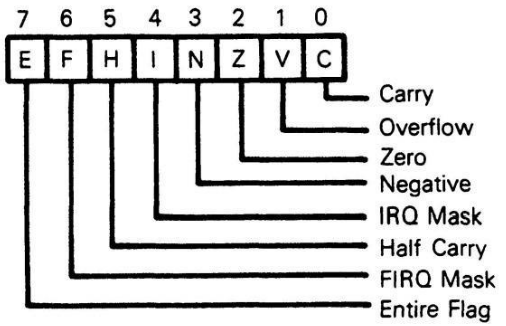


Figure 1-2. Condition Code Register

Scheduler

- The task scheduler takes action during <u>context changes</u>.
- It is the part of the kernel that is responsible for deciding the <u>next task to be performed</u> at any given time.
- The algorithm responsible for deciding what the next task to perform is called a <u>scheduling policy</u>.

Deadlines

For more info: Audsley, N.; Burns, A. (1990). Real-Time System Scheduling [PDF1, PDF2, PDF3] (Technical report). University of York, UK.

Deadline Characteristics

- A process can be divided into
  - Periodic
  - Aperiodic
- A periodic processes can be characterized by
  - Its period (**T**)
  - Its required execution time (per period)
- An aperiodic process can be characterized by:
  - Its activation distribution (a Poisson distribution e.g.)
  - Its required execution time (per event)

Deadline Characteristics

- Between the invocation and its deadline (**D**), the process requires a certain amount of computation time (**C**).
- Computation time may be static or vary within a given maximum and minimum.
  - Therefore

$$C \leq D$$

  - Should hold true for all processes.

Deadline Characteristics

- For periodic processes, the period must be at least equal to its deadline.
- This is known as *runnability constraint*

$$C \leq D \leq T$$

Type of deadlines

- The **requirement** of a routine/process/task that needs to be executed with a given deadline
- The task result is **only worth** something inside before deadline
- If the process finish before the start time (for a periodic task) or after the deadline the result can be discarded

Type of deadlines

Hard deadline

- In a real system, the situation may even be worse:
  - A missed deadline can bring damage to the system
- In those type of systems, the longer the waiting the bigger the damage.

Type of deadlines

Soft deadline

- For most real systems, not all tasks/events have hard or critical deadlines.
- A soft deadline is the one that missing it, may reduce the outcome value, but does not translate to a damage to the system

Type of deadlines

- A tasks can also have both a soft and a hard deadline

Type of deadlines

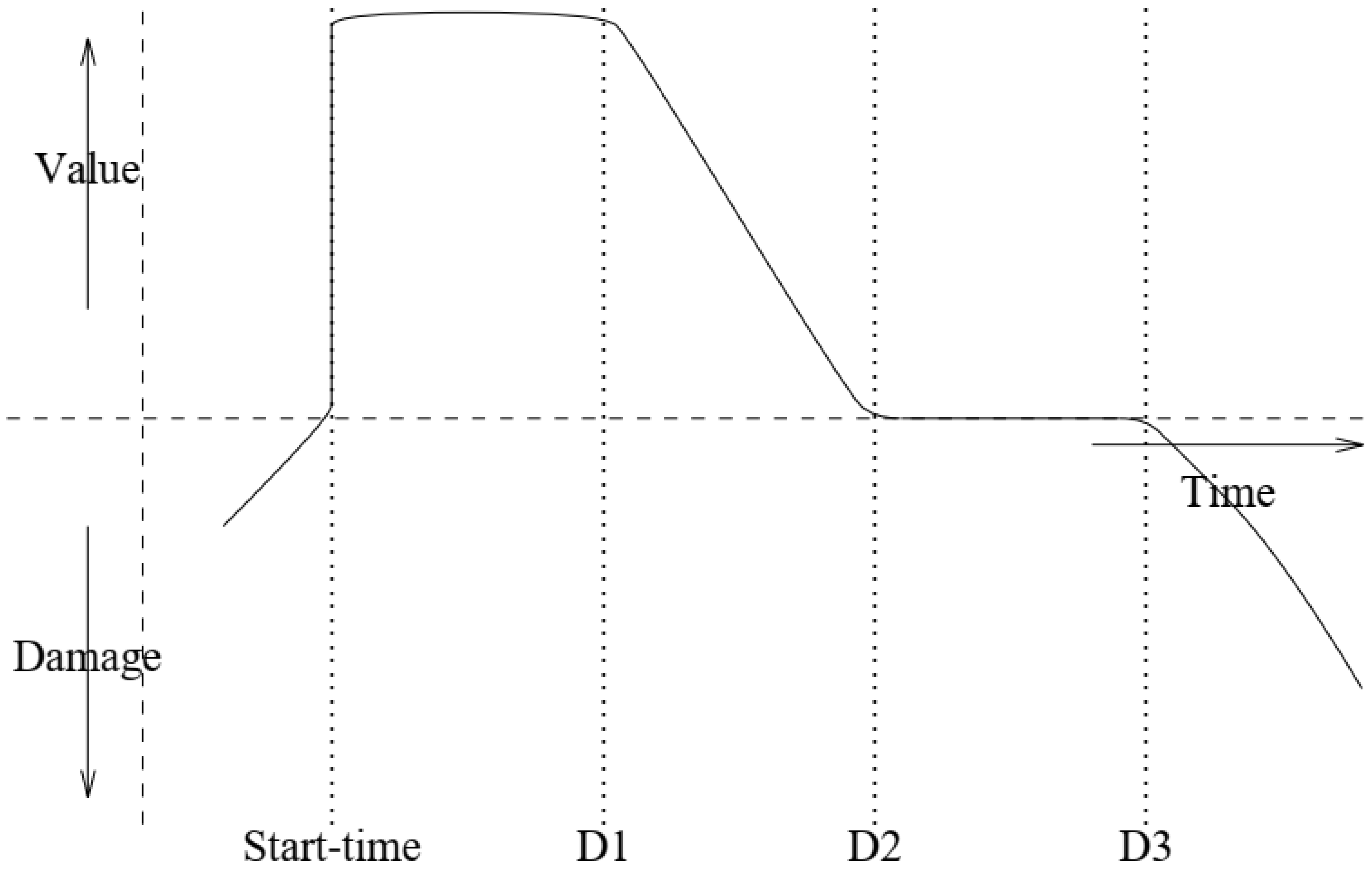


Figure 4: A Hybrid System

Process Scheduling

Process Scheduling

It must have an algorithm that cares about 5 rules:

- **Fairness** – All processes must have access to the CPU (timeout)
- **Efficiency** – seek maximum CPU utilization
- Minimize **Response Time**
- **Turnaround** – Minimizes batch users.Process completion time (allocation + queue + CPU execution + I/O execution)
- **Throughput** – Maximize the number of jobs processed

Process Scheduling

CPU scheduling can be classified into:

Non-preemptive:

- Simplest implementation of the scheduler.
- Processes uses the processor for as long as they want.
- The process leaves/releases the CPU under the following conditions:
  - End of execution; or
  - I/O operation request (voluntary).

Process Scheduling

CPU scheduling can be classified into:

Preemptive:

- More complex scheduler.
- Process uses the processor during a defined portion of time (**time-sharing**).
- CPU sharing is guaranteed. Periodically the scheduler interrupts the running process and switches it to the “ready” state.
- The process leaves/releases the CPU under the following conditions:
  - End of execution; or
  - I/O operation request; or
  - **Quantum end**.

Practical example

Practical example

- The only struct field is the function pointer. Other fields will be added latter.
- The circular buffer open a new possibility:
  - A process now can state if it wants <u>to be rescheduled</u> or if it is a <u>one-time run</u> process
  - In order to implement this <u>every process must return a code</u>.
  - This code also says if there was <u>any error</u> in the process execution

Practical example

- Cooperative kernel example
  - The presented code can be compiled in any C compiler
- The kernel consists of three functions:
  - KernelInit (): Initializes internal variables
  - KernelAddProc (): Adds processes in the pool
  - KernelLoop (): Initializes the process manager
    - This function has an infinite loop because it only needs to be terminated when the equipment / board is switched off.

Practical example

```
//return code
#define SUCCESS    0
#define FAIL       1
#define REPEAT     2

//function pointer declaration
typedef char(*ptrFunc)(void);

//process struct
typedef struct {
   ptrFunc func;
} process;

Process * pool[POOL_SIZE];
```

Practical example

```c
char kernelInit(void){
    start = 0;
    end = 0;
    return SUCCESS;
}
char kernelAddProc(process * newProc){
    //checking for free space
    if ( ((end+1)%POOL_SIZE) != start){
        pool[end] = newProc;
        end = (end+1)%POOL_SIZE;
        return SUCCESS;
    }
    return FAIL;
}
```

Practical example

```c
void kernelLoop(void){
  for(;;){
      //Do we have any process to execute?
      if (start != end){
        //check if there is need to reschedule
        if (pool[start]->func() == REPEAT){
          kernelAddProc(pool[start]);
        }
        //prepare to get the next process;
        start = (start+1)%POOL_SIZE;
      }
   }
}
```

Practical example

- Presenting the new processes

```
char tst1(void){
    printf("Process 1\n");
    return REPEAT;
}
char tst2(void){
    printf("Process 2\n");
    return SUCCESS;
}
char tst3(void){
    printf("Process 3\n");
    return REPEAT;
}
```

Practical example

```c
void main(void){
   //declaring the processes
   process p1 = {tst1};
   process p2 = {tst2};
   process p3 = {tst3};
   kernelInit();
   //Test if the process was added successfully
   if (kernelAddProc(&p1) == SUCCESS){
      printf("1st process added\n");}
   if (kernelAddProc(&p2) == SUCCESS){
      printf("2nd process added\n");}
   if (kernelAddProc(&p3) == SUCCESS){
      printf("3rd process added\n");}
   kernelLoop();
}
```

Practical example

```c
void kernelLoop(void){
   for(;;){
      //Do we have any process to execute?
      if (start != end){
         //check if there is need to reschedule
         if (pool[start]->func() == REPEAT){
            kernelAddProc(pool[start]);
         }
         //prepare to get the next process;
         start = (start+1)%POOL_SIZE;
      }
   }
}
```

7-13 programming code lines
Circular buffer
Function pointer
Main Loop executes the process
In order of inclusion
Interrupts can add process

Practical example
Output

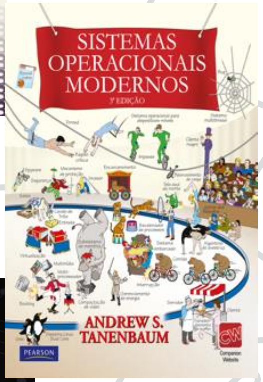
```
Console Output:
--------------------------
1st process added
2nd process added
3rd process added
Ite. 0, Slot. 0: Process 1
Ite. 1, Slot. 1: Process 2
Ite. 2, Slot. 2: Process 3
Ite. 3, Slot. 3: Process 1
Ite. 4, Slot. 0: Process 3
Ite. 5, Slot. 1: Process 1
Ite. 6, Slot. 2: Process 3
Ite. 7, Slot. 3: Process 1
Ite. 8, Slot. 0: Process 3
...
--------------------------
```

Bibliography

- Denardin, G. B.; Barriquello, C. H. **Sistemas operacionais de tempo real e sua aplicação em sistemas embarcados**. 1ª ed. Editora Blucher. ISBN: 9788521213970. https://plataforma.bvirtual.com.br/Acervo/Publicacao/169968

- Tanenbaum, A.S. **Sistemas Operacionais Modernos**. 3ª ed. 674 páginas. São Paulo: Pearson. ISBN: 9788576052371.
  https://plataforma.bvirtual.com.br/Acervo/Publicacao/1233

- Almeida, Moraes, Seraphim e Gomes. **Programação de Sistemas Embarcados**. 2ª ed. Editora GEN LTC. ISBN: 9788595159105.
  https://cengagebrasil.vitalsource.com/books/9788595159112



Available at: https://unifei.edu.br/ensino/bibliotecas/

Embedded Operating Systems

Prof. Otávio Gomes
otavio.gomes@unifei.edu.br

in /otavio-gomes