The University of Texas at Austin
Department of Electrical and Computer Engineering

Real-Time Systems / Real-Time Operating Systems

EE445M/EE380L.12, Spring 2019

Final Exam Solutions

Date: May 20, 2019

UT EID: ______________

Printed Name: ____
Last, First

Your signature is your promise that you have not cheated and will not cheat on this exam, nor will you help others to cheat on this exam:

Signature: ____

Instructions:

- Open book and open notes.
- No calculators or any electronic devices (turn cell phones off).
- Please be sure that your answers to all questions (and all supporting work that is required) are contained in the space (boxes) provided.
- *Anything outside the boxes will be ignored in grading.*
- For all questions, unless otherwise stated, find the most efficient (time, resources) solution.

Problem 1	10	
Problem 2	15	
Problem 3	15	
Problem 4	15	
Problem 5	25	
Problem 6	20	
Total	100	

Problem 1 (10 points): Synchronization

Consider an OS that manages three resources. Threads can require two out of three resources, but never all three. Given below is the code for an agent thread running as part of the operating system handing out resources to different user threads (where multiple instances of each type can be running) according to an OS-internal policy. User and agent threads synchronize using semaphores *s1*, *s2* and *s3* associated with each resource as well as a *done* semaphore indicating that user threads are finished using resources. All semaphores are initialized to zero.

```
OS_Agent() {
  while(1) {
    switch(policy()) {
      case 1:
        ~~OS_bSignal(&s1);~~   OS_Signal(&s12);
        ~~OS_bSignal(&s2);~~
        break;
      case 2:
        ~~OS_bSignal(&s2);~~   OS_Signal(&s23);
        ~~OS_bSignal(&s3);~~
        break;
      case 3:
        ~~OS_bSignal(&s1);~~   OS_Signal(&s13);
        ~~OS_bSignal(&s3);~~
        break;
    }
    OS_bWait(&done);
  }
}
```

```
Thread1() {
 while(1) {
  OS_bWait(&s12);
  ~~OS_bWait(&s2);~~
  …
  OS_bSignal(&done);
 }
}
```

```
Thread2() {
 while(1) {
  OS_bWait(&s23);
  ~~OS_bWait(&s3);~~
  …
  OS_bSignal(&done);
 }
}
```

```
Thread3() {
 while(1) {
  OS_bWait(&s13);
  ~~OS_bWait(&s3);~~
  …
  OS_bSignal(&done);
 }
}
```

a) What is wrong with this code?

Deadlock. E.g. assume agent notifies s1 and s2, then Thread1 grabs s1 and Thread2 grabs s2.

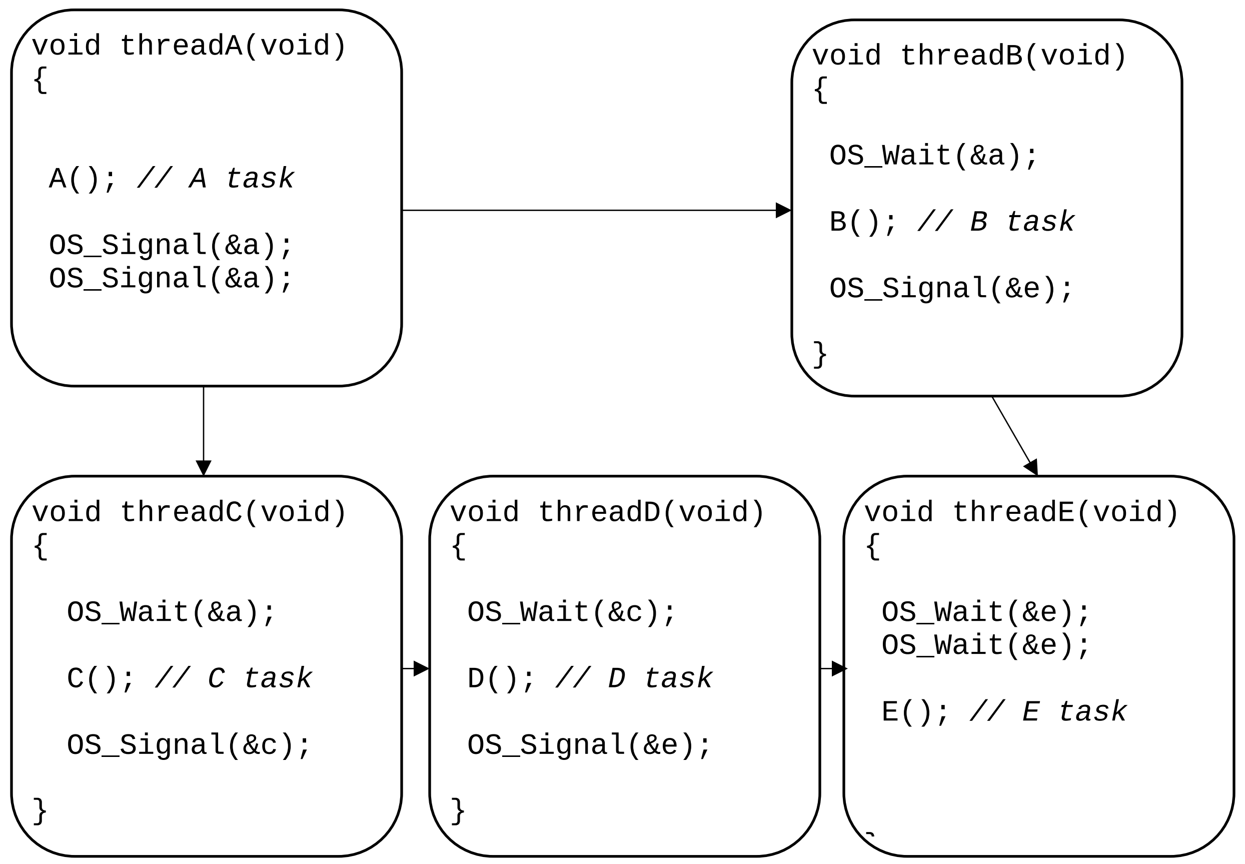
b) Can the code be modified to fix the problem while keeping the OS general, i.e. without hardcoding the number of instances running of each user thread type and without adding new threads? If so, modify the code accordingly. If not, why not?

Combine semaphores into one semaphore per thread type.
-> This is the classic Cigarette Smokers Problem. There are other variants and solutions. See the little Book of Semaphores.

Problem 2 (15 points): Scheduling

Given the task graph from Problem 3 in the midterm:

```
// Global variables and initialization code

sema_t a = 0;
sema_t c = 0;
sema_t e = 0;
```



```
void threadA(void)
{

 A(); // A task

 OS_Signal(&a);
 OS_Signal(&a);
```

```
void threadB(void)
{
 OS_Wait(&a);

 B(); // B task

 OS_Signal(&e);

}
```

```
void threadC(void)
{

  OS_Wait(&a);

  C(); // C task

  OS_Signal(&c);

}
```

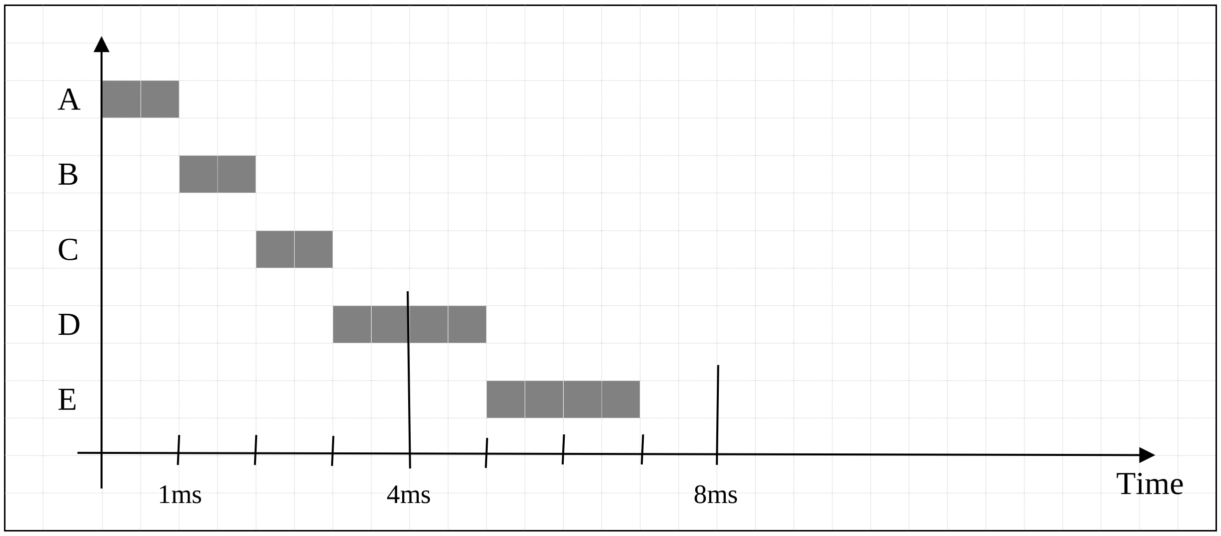
```
void threadD(void)
{

 OS_Wait(&c);

 D(); // D task

 OS_Signal(&e);

}
```

```
void threadE(void)
{

 OS_Wait(&e);
 OS_Wait(&e);

 E(); // E task
```

Now assume that we run this task graph on your perfectly functioning OS from Lab 3 using a priority scheduler. Further assume that the tasks have the following execution times and deadlines (relative to time 0).

Task	**Execution Time**	**Deadline**	**Priority**
A	1ms	5ms	*4*
B	1ms	6ms	*3*
C	1ms	7ms	*2*
D	2ms	4ms	*5 (highest)*
E	2ms	8ms	*1 (lowest)*

a) Assign priorities to threads to realize a strict earliest deadline first (EDF) scheduling. Assuming all threads are launched at time 0, show the schedule of task executions.



b) Are any of the deadlines violated? If so, is there an alternative priority assignment of priorities that would result in a valid schedule without deadline violations (show the priorities and change in the schedule)? If not, will EDF always be able to find a valid scheduler for a task graph with dependencies, if one exists (show why)?

D violates its deadline. Alternative priority assignment and schedule requires C and D to run before B, e.g. A(5), C(4), D(3), B(2), E(1).

EDF will not always find a valid schedule for tasks with dependencies. But it can be extended by adjusting each tasks' deadline to be the smallest of its original deadline and the deadlines minus execution times of all its dependents, and doing so recursively / from the end of the chain backwards, where priorities are then assigned as in normal EDF (i.e. earliest adjusted deadline first). In this example, adjusted task deadlines would be:

A: min(5ms, 6ms-1ms, 2ms-1ms) = 1ms
B: min(6ms, 8ms-2ms) = 6ms
C: min(7ms, 4ms-2ms) = 2ms
D: min(4ms, 8ms-2ms) = 4ms
E: 8ms

*This variant of EDF that can handle task graphs with dependencies is called EDF**

c) Is there any priority inversion in your schedule from a)? If so, would a priority inheritance or priority ceiling protocol fix the inversion including any deadline violations (if any)?

Technically, there is inversion between A,B,C that run before D, which has an earlier deadline than all of them. In EDF, this is sometimes also called deadline interchange.

Priority ceiling/inheritance will not fix inversion between A/C and D. It is inherent in the dependencies. But priority inheritance can replicate deadline adjustments above (by defining predecessor tasks to be semaphore holders and raising their priority to the one of the successor) such that inversion between B and C,D and deadline violations are avoided.

Note that priority ceiling/inheritance also don't fix inversions inherent in mutual exclusion between two tasks in regular periodic systems. The main problem they solve are unbounded inversions from unrelated tasks with intermediate priorities in between.

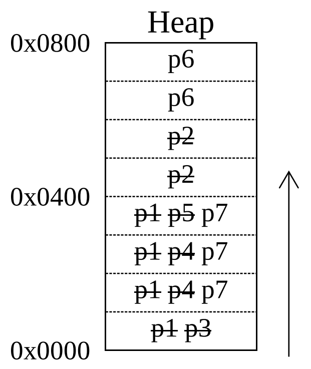
Problem 3 (15 points): Heap Management

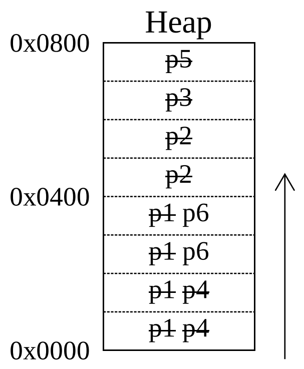
Consider a heap of size 2048 bytes. Assuming an initially empty heap, given the following sequence of heap allocations and de-allocations:

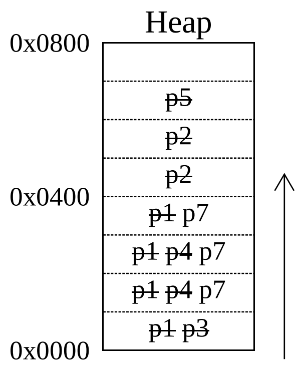
```
p1 = Heap_Malloc(1024);
p2 = Heap_Malloc(512);
Heap_Free(p1);
p3 = Heap_Malloc(256);
p4 = Heap_Malloc(512);
p5 = Heap_Malloc(256);
p6 = Heap_Malloc(512);
Heap_Free(p4);
Heap_Free(p5);
p7 = Heap_Malloc(768);
Heap_Free(p2);
Heap_Free(p3);
```

Show the state of the heap at the end of this sequence using a first fit, best fit and worst fit allocation scheme. Assume that the heap manager does not require any overhead for extra meta-data, and that the heap is allocated from bottom to top, i.e. a block always ends up being placed at the bottom (lowest address) of its chosen free space. If an allocation fails, assume that the function returns a null pointer and the rest of the sequence continues. For each allocation scheme, answer the following questions: (1) Does the sequence complete without error? If not, show the allocations that fail. (2) What is the amount of free space and what is the largest block size that can be allocated after this sequence?

a) First fit b) Best fit c) Worst fit







Sequence completes without error. *Free space: 768* *Largest allocatable: 512*	*p7 allocation fails.* *Free space: 1536* *Largest allocatable: 1024*	*p6 allocation fails* *Free space: 1280* *Largest allocatable: 1024*

Problem 4 (15 points): Process Management

Recall that in Lab 5, you were given an ELF loader to load a process from the SD card. The ELF loader's *exec_elf()* routine parses the ELF file, loads its code and data segments into dynamically allocated memory regions on the heap and finally calls *OS_AddProcess()* as shown below:

```
int exec_elf(const char *path, const ELFEnv_t *env)
{ …
  f_open(&f, path, FA_READ);       // open & read ELF header
  …

  text = Heap_Malloc(code_size);   // allocate & load code segment
  f_read(f, text, code_size);


  data = Heap_Malloc(data_size);   // allocate & load data segment
  f_read(f, data, data_size);


  …                                // relocation using 'env'

  f_close(f);
  return OS_AddProcess(entry, text, data);  // add OS process
}
```

a) Given basic (but incomplete) implementations of *OS_AddProcess()* and *OS_Kill()* below, complete the code to properly manage process memory and reclaim it when the process exits.

```
struct PCB {
  unsigned long Id;
  unsigned long numThreads;

  void *text;
  void *data;

};
struct PCB pcbs[NUMPCB];

struct TCB {
  struct TCB* next;
  unsigned int* sp;
  unsigned long Id;
  struct PCB *process;




};
struct TCB tcbs[NUMTCB];
struct TCB *RunPt;
```

```
int OS_AddProcess(void(*entry)(void), void *text, void *data){
  int sr, n;
  EnterCritical()
  for(n = 0; n < NUMPCB; n++) {
    if(pcbs[n].Id == 0) break; // found it; Id = 0 implies free PCB
  }
  if(n == NUMPCB) { ExitCritical(); return 0; }

  pcbs[n].Id = ++processId;
  pcbs[n].numThreads = 1;


  pcbs[n].text = text;
  pcbs[n].data = data;




  ExitCritical()
  return OS_AddThread(entry, STACK_SIZE, &pcb[n]);
}                                     // sets TCB.process = &pcb[n]


void OS_Kill(void) {
  struct TCB *p;
  DisableInterrupts();

  if(RunPt->process) {
    RunPt->process->numThreads--;



    if(RunPt->process->numThreads == 0)
    {
      RunPt->process->Id = 0;    // mark as free

      Heap_Free(RunPt->process->text);
      Heap_Free(RunPt->process->data);

    }


  }



  // Remove thread and return to TCB pool
  p = RunPt;
  while((p->Next) != RunPt) p = p->Next;
  p->Next = RunPt->next;
  RunPt->Id = 0;       // mark as free

  ContextSwitch();
  EnableInterrupts();
}
```

b) Assuming a program on disk with a code segment of size 128 bytes and a data segment of size 384 bytes. Given a heap size of 4096 bytes and assuming that the heap is not used for anything else, what is the maximum number of concurrent instances of this program that you can load and run?

4096 / (128+384) = 8 instances.

c) Is it possible to increase the number of concurrent instances of the same program that can be loaded and running? If not, why not? If yes, describe how this can be done and sketch the changes that would need to be made to the ELF loader or OS code shown above. What is the maximum number of concurrent instances in this case?

Code segment can be shared among all instances, i.e. only needs to be loaded once (when the first instance is loaded).

(4096 – 128) / 384 = 10 instances.

Code changes needed:

1) Extend PCB to remember which program it belongs to, e.g. by storing the corresponding file path (passed from the ELF loader into OS_AddProcess):

```
 int OS_AddProcess(…, const char* path){
    …
    pcbs[n].path = path;
```

2) Extend ELF loader to check whether program is already loaded and only allocate new code segment if not, otherwise reuse existing:

```
  int exec_elf(const char *path, …) {
    …
    for(n = 0; n < NUMPCB; n++)
      if(pcbs[n].Id && strcmp(path, pcbs[n].path)) break;
    if(n == NUMPCB) {
      text = Heap_Malloc(code_size);
      f_read(f, text, code_size);
    } else {
      text = pcbs[n].text;
    }
```

3) Modify OS_Kill to release code segment only if there is no other PCB with same path:

```
  void OS_Kill(void) {
    …
      for(n = 0; n < NUMPCB; n++)
        if(pcbs[n].Id && strcmp(RunPt->process->path, pcbs[n].path))
          break;
      if(n == NUMPCB) {
        Heap_Free(RunPt->process->text);
      }
```

Problem 5 (25 points): File System

Assume two separate disks of identical size 4kB with 16 blocks each, where one disk uses a filesystem with indexed and the other a filesystem with linked allocation. Further assume that the exact same two files ‘A’ and ‘G’ (but nothing else) have been copied onto both disks.

a) Given the partial disk states (blocks, directory and file system tables) as shown below, complete the missing information on each disk to represent correct filesystems. Mark each block with the name of the file it belongs to (or ‘*’ if empty space). Assume that files ‘A’ and ‘G’ fit into the same number of blocks on both disks.

Indexed

Directory

File	Start	Blocks
A	0	3
G	*3*	7
*	*10*	*4*

* Free space

	Index Table
0	7
1	4
2	9
3	11
4	12
5	5
6	14
7	13
8	*15*
9	3
10	2
11	10
12	8
13	6
14	
15	

	Disk
0	Dir
1	Idx
2	*
3	G
4	A
5	G
6	*
7	A
8	*
9	A
10	*
11	G
12	G
13	G
14	G
15	G

Linked

Directory

File	Start	Blocks
A	4	*3*
G	*14*	*7*
*	*1*	*5*

* Free space

	Disk	
0	Dir	-
1	*	5
2	G	8
3	G	11
4	A	6
5	*	12
6	A	9
7	*	0
8	G	0
9	A	0
10	G	*3*
11	G	2
12	*	13
13	*	7
14	G	15
15	G	10

b) What is the space overhead of the two filesystems, i.e. how many out of the 4096 bytes on each of the two disks are usable for file data storage?

Indexed	Linked
Block size is 4096/16 = 256 bytes. *Overhead is two 256 byte blocks for directory and index:* *4096 – 512 = 3584 bytes usable* *Potentially, directory and index can be combined into one block:* *4096 – 256 = 3840 bytes usable*	*Block size is 4096/16 = 256 bytes.* *Overhead is one 256 byte blocks for directory and1 byte per remaining 15 blocks for links:* *4096 – 256 – 15 = 3825 bytes usable*

c) Given the sequence of file system calls below, show the order of disk accesses that are made by the each of the two filesystems. Assume that nothing is initially in memory. List the disk block numbers accessed (read or written). How many disk accesses are made in each case?

```
 f_open(&fa, “A”, FA_READ);    // open ‘A’
f_seek(fa, 306);     // read integer from byte position 306
f_read(fa, &x, 4);
f_seek(fa, 545);     // read integer from byte position 545
f_read(fa, &y, 4);
f_seek(fa, 319);     // read integer from byte position 319
f_read(fa, &z, 4);
f_close(fa);
gpa = (x + y + z) / 3;
f_open(&fg, “G”, FA_WRITE);
f_seek(fg, 21);          // write result to byte position 21
f_write(fg, &gpa, 4);
f_close(fg);
```

Indexed	Linked
Read directory *Read index table* *Read block 4 (306/256 = 2nd block of A)* *Read block 9 (545/256 = 3rd block of A)* *Read block 4 (319/256 = 2nd block of A)* *Read block 11 (1st block of G)* *Write back block 11* *6 reads and 1 write = 7 accesses*	*Read directory* *Read block 4 (1st block of A)* *Read block 6 (2nd block of A)* ~~*Read bock 4*~~ *(avoid if last block kept in mem)* ~~*Read block 6*~~ *Read block 9 (3rd block of A)* *Read block 4* *Read block 6 (2nd block of A)* *Read block 14 (1st block of G)* *Write back block 14* *7(9) reads and 1 write = 8(10) accesses*

d) Can the file system performance be improved, i.e. can the number of disk accesses be reduced in either case? If so, how, and how many disk accesses are made in that case?

Performance can be improved by including a disk cache that retains a number of blocks recently ready from disk.

With perfect caching (requiring space for 3 blocks beyond directory and index tables in this example), the number of accesses is reduced to 5 reads and 1 write = 6 accesses in both cases.

Problem 6 (20 points): Remote Procedure Call

You are asked to implement a remote procedure call (RPC) between two micro-controllers over the CAN bus. With RPCs, the OS provides the capability for a thread on one machine to call a function implemented on another machine while making it look to the programmer as if the thread is just calling a local function, where the OS on both machines transparently handle all the necessary communication and synchronization to execute the call remotely. Given the high-level CAN driver (`can0.h`) from the lab with driver primitives as listed below, complete the code such that on the RPC client machine can call function on the RPC server. Show any changes to the driver code that you need to make.

Client:

```
// *** can0.h ***
#define CAN_BITRATE 1000000
#define RCV_ID 2
#define XMT_ID 4

int CAN0_CheckMail(void);
int CAN0_GetMailNonBlock(uint8_t data[4]);
void CAN0_GetMail(uint8_t data[4]);
void CAN0_Open(void);
void CAN0_SendData(uint8_t data[4]);
```

```
// local function stub to call remote function
uint8_t PWM_Duty(uint16_t left, unit16_t right) {
  uint8_t data[4];

  data[0] = left & 0XFF;    // marshalling of data into a CAN packet
  data[1] = left >> 8;
  data[2] = right & 0xFF;
  data[3] = right >> 8;

  CAN0_SendData(data);      // send packet and receive response
  CAN0_GetMail(data);

  return data[0];           // De-marshalling of response
}

void speedThread(void) {
  …
  if(!PWM_Duty(l, r)) goto error;
  …
}

void main(void) {
  PLL_Init();
  OS_Init();

  CAN0_Open();

  OS_AddThread(&speedThread, 128);
  …
  OS_Launch(TIME_1MS);
}
```

*Name:*______________________________

Server:

```
// *** can0.h ***
#define CAN_BITRATE 1000000
#define RCV_ID 4
#define XMT_ID 2

int CAN0_CheckMail(void);
int CAN0_GetMailNonBlock(uint8_t data[4]);
void CAN0_GetMail(uint8_t data[4]);
void CAN0_Open(void);
void CAN0_SendData(uint8_t data[4]);
```

```
// local function
uint8_t PWM_Duty(uint16_t left, unit16_t right) {
  PWM_0_CMPA_R = left - 1;
  PWM_1_CMPA_R = right – 1;
  return motor_error;
}

// server thread
void RPC_Server(void)
{
  uint8_t  data[4];
  uint16_t left, right;
  uint8_t  res;

  while(1) {
    CAN0_GetMail(data);                    // Wait for message

    left = data[1]<<8 + data[0];           // De-marshall data
    right = data[3]<<8 + data[2];

    res = PWM_Duty(left, right);           // Call function

    data[0] = res;                         // Marshall & send response
    CAN0_SendData(data);
  }
}

void main(void)
{
  PLL_Init();
  OS_Init();
  PWM_Init();

  CAN0_Open();

  OS_AddThread(&RPC_Server, 128);
  …

  OS_Launch(TIME_1MS);
}
```