Digital Systems

Problem 1 – Consider an oscillating signal that comes from an input pin (push button) that needs to be debounced. When the push button is pressed, the signal is 1; when it is released the signal is 0. The oscillations occur immediately after the push button has been pressed or released. Design a digital system that measures the shortest duration when the signal is 0 and the shortest duration when the signal is 1. It is considered that the reference clock signal is 50 MHz (and hence you cannot have a precision finer than 20ns).

Problem 2 – Consider an oscillating signal that comes from an input pin (push button) that needs to be debounced. Design a digital system that implements debouncing, as well as it generates periodic single pulses as follows. After the push button has been pressed, the first single pulse is generated. Thereafter, so long as the push button continues to be pressed a single pulse will be generated each time a user-programmable interval has elapsed. Note, a single pulse is equal in duration to the reference clock period. It is considered that the reference clock signal is 50 MHz and that the push button is considered released if 0 has been measured for 10 ms.

Problem 3 – Consider two arrays A and B containing 16 bit signed numbers stored in one memory with 256 locations with 16 bits per location. Both arrays have 128 elements, with A[i] stored in location “i” and B[i] stored in location “128 + i”. Design a digital circuit that computes the average of the sum-of-absolute-differences (SAD), as follows. SAD equals the sum of the absolute values of A[i]-B[i] for “i” from 0 to 127. After computing SAD, the average value (on 16 bits) is available on the output. First assume that the memory is a single-port RAM with one clock cycle latency. Then repeat the same problem if the memory changes to a dual-port RAM. The aim is to reduce the number of clock cycles required to complete the calculation.

Data Structures in C

Problem 4 – Consider the enclosed C program. Assuming that characters are represented on 1 byte, short integers on 2 bytes, integers on 4 bytes, floats on 4 bytes and doubles on 8 bytes, show the messages that are printed when the program is executed.

```
#include <stdio.h>
#include <stdlib.h>

int main(void) {

     char text[] = "A short message for this problem";

     void *void_pointer = (void *)text;
     short int *short_int_pointer = (short int *)void_pointer;
     float *float_pointer = (float *)void_pointer;

     struct my_struct{
          int i;
          double d;
     };
     struct my_struct *my_struct_pointer = (struct my_struct *)void_pointer;

     union my_union{
          int i;
          double d;
     };
     union my_union *my_union_pointer = NULL;

     short_int_pointer += 5;
     float_pointer += 4;
     my_struct_pointer++;
     my_union_pointer = (union my_union *)my_struct_pointer - 1;

     printf("%s\n", (char *)void_pointer);
     printf("%s\n", (char *)short_int_pointer);
     printf("%s\n", (char *)float_pointer);
     printf("%s\n", (char *)my_struct_pointer);
     printf("%s\n", (char *)my_union_pointer);

     return 0;
}
```

Problem 5 – Consider the following piece of C code:

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_WORD_SIZE  10

char *extract_reversed_word(unsigned int word_index)
{
     char *message = "A short message for this problem";
...
}

int main(void)
{
     unsigned int i=0;
     printf("%s\n", extract_reversed_word(i));

     return 0;
}
```

Complete the “extract_reversed_word” function as follows. The “word_index” is used to identify a word from the message defined within the “extract_reversed_word” function. Then it returns it in reversed order. For this example, if “word_index” is 1 then the function returns a pointer to the character array “trohs”. For simplicity reasons, it is assumed that the words are separated only through spaces. We also assume that the function argument “word_index” is guaranteed not to exceed the maximum word count in the message and the maximum word size is defined in the file as MAX_WORD_SIZE. *Tip*: use pointers to static variables within the function.

Problem 6 – Consider the following three data structures used to represent three-dimensional arrays where each entry for each dimension has its own size.

```
struct vector {
     short int size;
     short int *vec;
};

struct vector2 {
     short int size;
     struct vector *vec;
};

struct vector3 {
     short int size;
     struct vector2 *vec;
};
```

Write a program that first initializes the sizes of each dimension randomly and allocates the memory dynamically (based on these sizes) before initializing the short integers to random values. Note, the random values (for both sizes and elements) are considered to be between 1 and a pre-defined MAX_VALUE. After the three-dimensional array has been initialized, compute its average and then de-allocate the memory before terminating the program.

Problem 7 – Consider the following two data structures.

```
struct my_struct {
     int i;
     float f;
};

union my_union {
     struct my_struct s;
     double d;
};
```

Assume that the user initializes a “union my_union” array first by storing random values in the “struct my_struct” field. Then it searches for the minimum and maximum values using:

```
void find_min_max  (union my_union my_union_array[], int (*my_union_compare),
                    union my_union *my_union_min, union my_union *my_union_max)
```

After min/max are printed, the union array is assigned new random values in the “double” field and the new min/max are printed. Note, the search function uses a generic comparison function:

```
int (*my_union_compare)(const void *, const void *);
```

Depending on which field of the union is used, the “my_union_compare” function needs to be updated with a custom comparison function. You can define your own criteria for comparison.

Organization of Integrated Embedded Systems

Problem 8 – Figure 1 shows a generic organization for embedded systems integrated into a programmable chip device. For the sake of simplicity, the instruction and data memories are not shown and their access is assumed to be identical as for a peripheral.

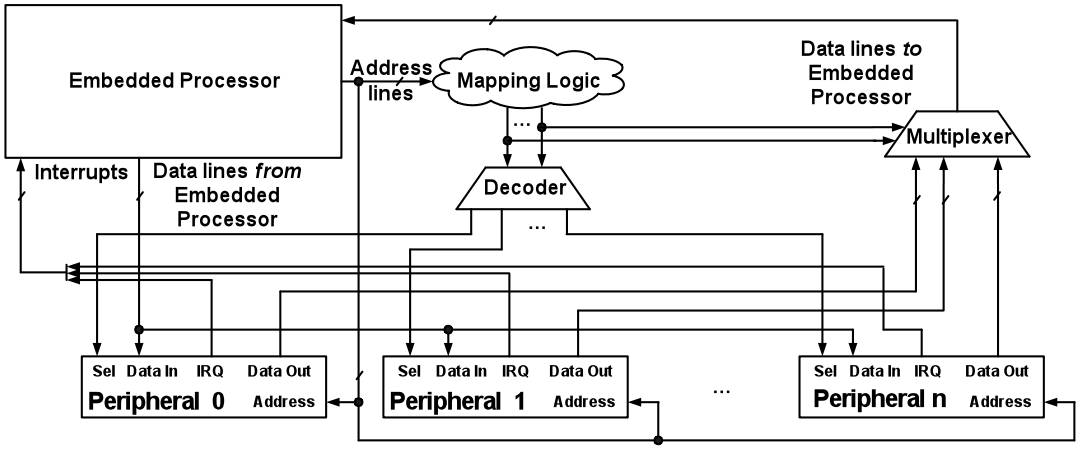


Figure 1 – An embedded processor and its generic interconnection to peripherals.

It is assumed that we have 9 peripherals connected to the embedded processor at the following addresses in hex format: peripheral 0 - 0x2310; peripheral 1 - 0x1240; peripheral 2 - 0x1250; peripheral 3 - 0x1260; peripheral 4 - 0x1270; peripheral 5 - 0x1280; peripheral 6 - 0x1290; peripheral 7 - 0x12A0; peripheral 8 - 0x12B0. Note, the address bus is on 16 bits and only the base addresses are given (and the 4 least significant bits of the address lines are used for local addressing within the peripheral). Derive and implement the mapping, decoding and multiplexing logic. Note, peripherals 1, 2, 5 and 8 are output-only peripherals, while all the others are either input-only or input/output peripherals.

Problem 9 – Consider an embedded system integrated as shown in Figure 1. We have 7 peripherals connected to the processor. If the mapping and decoding logic are as in Figure 2, then show the memory map of the system (the output 0 of the decoder is not connected).

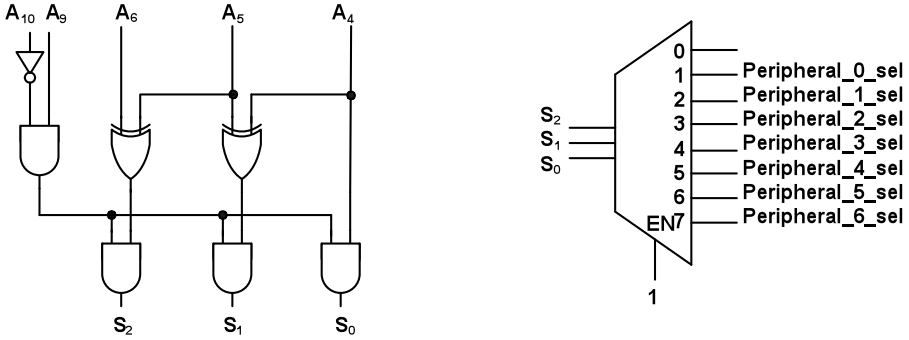


Figure 2 – Mapping and decoding logic for problem 9.

Note, the address bus is on 16 bits and only the base addresses are given (and the 4 least significant bits of the address lines are used for local addressing within the peripheral). Also, the address lines that are not used in the simplified implementation of the mapping logic are assumed to be 0 when computing the memory map.

Problem 10 – If the number of interrupt request (IRQ) lines for the embedded processor is only 4, suggest a solution to integrate 6 peripherals that generate interrupts in Figure 1.

Embedded Software

Problem 11 – Consider an embedded system as shown in Figure 4 from lab 2. If the number of inputs turned on from Group A is larger than the number of inputs turned off from Group A then turn on the least significant green light-emitting diode (LED). If the number of inputs turned on from Group B is more than the number of inputs turned off from Group B then turn on the most significant green LED. Write the software for this system without using interrupts.

Problem 12 – Solve problem 11 using interrupts.

Problem 13 – Consider an embedded system as shown in Figure 4 from lab 2. If an input from Group A has switched three times in a row (i.e., without any other inputs from Group A changing their position) then display in the terminal “Input # switched three times in a row”, where # is the input number. Write the software for this system without using interrupts.

Problem 14 – Solve problem 13 using interrupts.

Problem 15 – Consider an embedded system as shown in Figure 4 from lab 2. Whenever the least significant input from Group A is turned on then the least significant green LED will display the XOR of all the bits from Group B. If the least significant input from Group A is turned off then all the green LEDs will blink with turn on/off period equal (in milliseconds) to the value given by the configuration on the switches from Group B. For example, if the 9 switches in group B are 100000110, then all the green LEDs will be on for 262 ms; then off for 262 ms; then again on for 262 ms, ... Write the software for this system without using interrupts.

Problem 16 – Solve problem 15 using interrupts.

Problem 17 – Consider an embedded system as shown in Figure 4 from lab 2. There are 3 input sensors: *e* (enable), *w* (window) and *d* (door). There is 1 output called *s* (alarm sound). Input *e* is connected to the least significant switch from Group A; inputs *w* and *d* inputs are connected to the two least significant switches from Group B; *s* is connected to the least significant green LED. On power-up the system is disarmed. The system is disarmed until *e* is switched on. When the system is armed, if *e* is switched off then the system will be disarmed. If however, either *w* or *d* are switched on while the system is armed, signal *s* will be turned on for 900 ms; then it will turned off for 600 ms. This periodic sequence repeats until *e* is switched off, which disarms the system. Write the software for this system without using interrupts.

Problem 18 – Solve problem 17 using interrupts.

Problem 19 – Consider an embedded system as shown in Figure 4 from lab 2. This setup can be used to track the temperature in a room and activate/deactivate the heater, as explained below. Group A is connected to a temperature sensor on 9 bits. The least significant bit of Group B is connected to a periodic signal that switches every second. Whenever this periodic signal switches, a sample on 9 bits is recorded from Group A. If 30 consecutive samples are all below a minimum value (9’d18) the heater will be activated (the heater is activated by turning on the least significant bit of the green LEDs). The heater will stay active until 30 consecutive samples are all above a maximum value (9’d23). At this time it will be deactivated and it will stay as such until 30 consecutive samples are again all below 9’d18. It is assumed that on power up the heater is not active. Write the software for this system without using interrupts.

Problem 20 – Solve problem 19 using interrupts.

Real-time operating systems and scheduling

Problem 21 – Consider a real-time operating system (OS) that has a built-in preemptive scheduler. Each task has a unique priority and the lower the priority id, the higher the priority of the task. The time required to process interrupts is ignored and the time is in terms of OS ticks.

For this particular problem we have the following assumptions:

- periodic tasks are characterized as (*e*, *p*), where *e* is the execution time and *p* is the period between two consecutive arrival times for the respective task; for example, if a task with (*e*, *p*) arrives at time *t* then its next arrival time is at *t*+*p*
- a task that arrives at time *t* can be scheduled at any time between *t* and *t*+*p*-1 (failure to schedule it in this interval will mean that the input data will not be processed for this period)
- the first arrival time for all the periodic tasks in this problem is at OS tick 0

Consider the following three tasks scheduled statically (i.e., the priorities do not change): task A with priority 1 and (2, 10), task B with priority 2 and (3, 11) and task C with priority 3 and (4, 12).

(a) Show the schedule for a single-processor system for the first 20 OS ticks
(b) Show the schedule for a two-processor system (CPU1 & CPU2) for the first 20 OS ticks
(c) Repeat (b) with the following additions to the task list: task D with priority 4 and (5, 11) and task E with priority 5 and (5,12)

Note1: For each OS tick write which task is running (if the processor is idle, write X).
Note2: For a multi-processor system we have the following assumptions:

- if a task of higher priority arrives when all the processors are busy, it will be scheduled on the processor on which the task with the lowest priority is running (which will be preempted)
- if one or more processors are free, then the task will be scheduled on the processor on which it ran last time, provided that it is free; otherwise it will be scheduled on the idle processor with the lowest id (i.e., CPU1 has a lower id than CPU2)
- if two or more tasks arrive at the same OS tick, the OS will attempt to schedule all these tasks in the same OS tick; however the tasks with higher priority will be scheduled first
- if a task was preempted by a higher priority task, it will try to resume its execution on the processor on which it was preempted, if it becomes available first; otherwise it will be scheduled on any processor that becomes available first (in case of more than one idle processor, the one with the lowest id will be chosen)

Problem 22 – Consider the same setup as in problem 21, where the OS scheduler has been changed by the user to implement the non-preemptive scheduling policy. Address questions (a), (b) and (c) from problem 21.

Problem 23 – Consider the same setup as in problem 21, where the parameters for tasks A, B and C for questions (a) and (b) and tasks D and E for question (c) have changed as follows: task A has priority 5 and (3, 9), task B has priority 4 and (2, 10), task C has priority 3 and (2, 11), task D has priority 2 and (4, 9) and task E has priority 1 and (5, 11). Address questions (a), (b) and (c) from problem 21.

Problem 24 – Consider the same setup as in problem 23, where the OS scheduler has been changed by the user to implement the non-preemptive scheduling policy. Address questions (a), (b) and (c) from problem 23 (which are inherited from problem 21).

Problem 25 – Consider a real-time OS that has a built-in preemptive scheduler. Each task has a unique priority and the lower the priority id, the higher the priority of the task. The time required to process interrupts is ignored and the time is in terms of OS ticks.

For this particular problem we have the following assumptions:

- periodic tasks are characterized as (*e*, *i*), where *e* is the execution time and *i* is the idle time after the task has completed its execution; for example, if a task with (*e*, *i*) arrives at time *t*, then the next arrival time will be at *c*+*i*, where the *c* is the OS tick after the task has completed its execution
- the first arrival time for all the periodic tasks in this problem is at OS tick 0

Consider the following three tasks scheduled statically (i.e., the priorities do not change): task A with priority 1 and (2, 6), task B with priority 2 and (3, 6) and task C with priority 3 and (4, 6).

(a) Show the schedule for a single-processor system for the first 20 OS ticks
(b) Show the schedule for a two-processor system (CPU1 & CPU2) for the first 20 OS ticks
(c) Repeat (b) with the following additions to the task list: task D with priority 4 and (5, 6) and task E with priority 5 and (6,6)

Note1: For each OS tick write which task is running (if the processor is idle, write X).
Note2: For a multi-processor system we have the same assumptions as in problem 21.

Problem 26 – Consider the same setup as in problem 25, where the OS scheduler has been changed by the user to implement the non-preemptive scheduling policy. Address questions (a), (b) and (c) from problem 25.

Problem 27 – Consider the same setup as in problem 25, where the parameters for tasks A, B and C for questions (a) and (b) and tasks D and E for question (c) have changed as follows: task A has priority 5 and (3, 6), task B has priority 4 and (2, 8), task C has priority 3 and (2, 9), task D has priority 2 and (4, 5) and task E has priority 1 and (5, 6). Address questions (a), (b) and (c) from problem 25.

Problem 28 – Consider the same setup as in problem 27, where the OS scheduler has been changed by the user to implement the non-preemptive scheduling policy. Address questions (a), (b) and (c) from problem 27 (which are inherited from problem 25).

Problem 29 – Consider the same setup as in problem 25, where the tasks can be characterized by the following parameters (*e1*, *i1, e2, i2*), where *e1* and *e2* are execution times and *i1* and *i2* are idle times. Note, after the task ran for *e1* OS ticks, it will release the processor for *i1* OS ticks at which time it will wait for the scheduler to allow it to run again for *e2* OS ticks. The next arrival time will be *c*+*i2* where *c* is the OS tick after the task has completed the execution of the code section characterized by *e2*. The parameters for tasks A, B and C for questions (a) and (b) and tasks D and E for question (c) have changed as follows: task A has priority 1 and (1, 4, 1, 2), task B has priority 2 and (1, 3, 1, 3), task C has priority 3 and (2, 2, 2, 4), task D has priority 4 and (1, 2, 2, 3) and task E has priority 5 and (2, 3, 3, 3). Address questions (a), (b) and (c) from problem 25.

Problem 30 – Consider the same setup as in problem 29, where the OS scheduler has been changed by the user to implement the non-preemptive scheduling policy. Address questions (a), (b) and (c) from problem 29 (which are inherited from problem 25).

Problem 31 – Consider a real-time OS that has a built-in preemptive scheduler. Each task has a unique priority and the lower the priority id, the higher the priority of the task. The time required to process interrupts is ignored and the time is in terms of OS ticks.

For this particular problem we have the following assumptions:

- *aperiodic* tasks are characterized as (*e*, *d*), where *e* is the execution time and *d* is the relative deadline for completion after the arrival of the task; for example, if a task with (*e*, *d*) arrives at time *t*, then the OS tick when the task must complete is not later than *t+d*
- a custom scheduler task (called task S) runs for one OS tick only with priority 0 each time a new task arrives and its purpose is to create the task and register it in the OS and update *dynamically* the priorities of all the tasks that are currently registered with the OS; note the custom scheduler does not run after the completion of a task and it is assumed that a task deletes itself as soon as it has completed executing its code (note, the custom scheduler task does not delete itself)
- the dynamic priority assignment is done as follows: each time the custom scheduler is called it determines which active task was created first; the earlier its creation time, the higher its priority; it is assumed that two tasks are never created at the same time; note, this dynamic priority assignment policy is referred to as *first-in first-out (FIFO)*

We have the following three aperiodic tasks that have the following characteristics: task A with (5, 10) and arrival time 0; task B with (2, 7) and arrival time 2 and task C with (3, 8) and arrival time 4. Show the schedule for a single-processor system for the first 15 OS ticks. For each OS tick write which task is running (if the processor is idle, write X).

Problem 32 – Solve problem 31, with the following change in dynamic priority assignment.

- each time the custom scheduler is called it determines which active task was created last; the later its creation time, the higher its priority; it is assumed that two tasks are never created at the same time; note, this dynamic priority assignment policy is referred to as *last-in first-out (LIFO)*

Problem 33 – Solve problem 31, with the following change in dynamic priority assignment.

- each time the custom scheduler is called it determines which active task has the earliest deadline; the earlier the deadline of a task, the higher its priority; if two tasks have the same deadline, then the one that was created first will have higher priority and for the sake of simplicity it is assumed that if two tasks are created at the same time then their deadlines will be different; note, this dynamic priority assignment policy is referred to as *earliest deadline first (EDF)*

Problem 34 – Solve problem 31, with the following change in dynamic priority assignment.

- each time the custom scheduler is called it determines which active task has least slack to completion; the less slack a task has, the higher its priority; the slack of a task is computed as the difference between the deadline of the task and the sum between the current OS tick and how many OS ticks are still necessary to complete the execution of the task; if two tasks have the same slack then the one that was created first will have higher priority and for the sake of simplicity it is assumed that if two tasks are created at the same time then their slacks will be different; note, this dynamic priority assignment policy is referred to as *least slack time (LST)*

Problem 35 – Consider a real-time OS that has a built-in preemptive scheduler. Each task has a unique priority and the lower the priority id, the higher the priority of the task. The time required to process interrupts is ignored and the time is in terms of OS ticks.

For this particular problem we have the following assumptions:

- *aperiodic* tasks are characterized as (*e*, *i*), where *e* is the execution time and *i* is the idle time after the task has completed its execution; the task will subsequently be put on hold until the next arrival time is signaled through an asynchronous interrupt, at which time the task will be placed in the list of tasks that are ready to be executed
- there is no custom scheduler for this problem; rather the priorities are assigned statically before the tasks are created and the only way how the priority of task can be updated is when a task holds a mutex and it inherits the mutex priority (as explained next)
- for access to a shared resource, a mutex is employed; if a task locks a mutex, then it holds the mutex until the task completes its execution; only when a task of higher priority than the one that holds the mutex attempts to access the shared resource, the task who holds the mutex will automatically upgrade its priority (to the level defined when creating the mutex) and it will run with this priority until it releases the mutex; please note, if the task that holds the mutex is preempted by a task that does not require the mutex then when another task of a higher priority attempts to access the mutex, the task that holds the mutex will automatically update its priority to complete and release the mutex as soon as possible;
- for the sake of simplicity it is assumed that a task that competes for a shared resource will check for the mutex just before it is scheduled for execution and no code is executed after it releases the mutex; this simplifying assumption implies that if a task of higher priority than the one that holds the mutex is ready to be scheduled, then it will not run even a single OS tick until the mutex is released to it
- if two or more tasks compete for the mutex in the same OS tick then the one with the highest priority will obtain (and not the one that tried to access it first)

When the mutex is created its priority is set to 0. We have the following three aperiodic tasks with the following characteristics: task A with priority 3 and (5, 0) and arrival time 0; task B with priority 2 and (4, 0) and arrival time 2 and task C with priority 1 and (3, 0) and arrival time 4. For each of the following questions, show the schedule for a single-processor system for the first 15 OS ticks. For each OS tick write which task is running (if the processor is idle, write X).

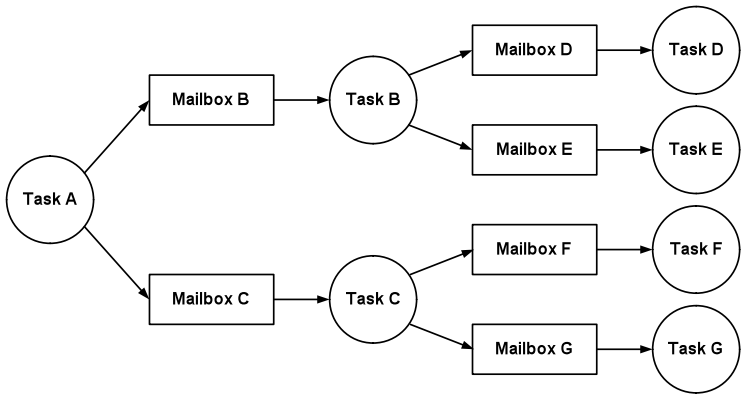
(a) all the tasks A, B and C compete for the mutex
(b) tasks A and B compete for the mutex, while task C does not require it
(c) tasks A and C compete for the mutex, while task B does not require it

Problem 36 – Consider the same setup as in problem 35, where the tasks have the following characteristics: task A with priority 1 and (3, 0) and arrival time 0; task B with priority 2 and (4, 0) and arrival time 2 and task C with priority 3 and (5, 0) and arrival time 4. Address questions (a), (b) and (c) from problem 35.

Problem 37 – Consider the same setup as in problem 35, with the following change to the way how the mutex priority is used. Unlike problem 35, where the priority of the task that holds the mutex is raised only when a task of higher priority attempts to access the shared resource, in this problem the priority of the task that holds the mutex is raised to the mutex priority as soon as it acquires the mutex after it is scheduled for execution. Address questions (a), (b) and (c) from problem 35.

Problem 38 – Consider a real-time OS that has a built-in preemptive scheduler. Each task has a unique priority and the lower the priority id, the higher the priority of the task. The time required to process interrupts is ignored and the time is in terms of OS ticks.

Consider the following task dependency graph.



For this particular problem we have the following assumptions:

- periodic tasks are characterized as (*e*, *i*), where *e* is the execution time and *i* is the idle time after the task has completed its execution
- a task with an input mailbox is suspended until a message is posted in its mailbox

Consider seven tasks with the following characteristics: task A with priority 1 and (2, 20); task B with priority 2 and (4, 0); task C with priority 3 and (3, 0); task D with priority 4 and (3, 0); task E with priority 5 and (2, 0); task F with priority 6 and (4, 0); task G with priority 7 and (2, 0). All of the above tasks are created at time 0 and their priorities are assigned statically.

For each of the following questions, show the schedule for a single-processor system for the first 20 OS ticks. For each OS tick write which task is running (if the processor is idle, write X).

(a) a message is posted in a mailbox after the source task has completed its execution
(b) a message is posted in a mailbox after the source task has executed for 1 OS tick
(c) a message is posted in a mailbox after the source task has executed for 2 OS ticks

Problem 39 – Consider the same setup as in problem 38, where the tasks have the following characteristics: task A with priority 7 and (2, 20); task B with priority 6 and (4, 0); task C with priority 5 and (3, 0); task D with priority 4 and (3, 0); task E with priority 3 and (2, 0); task F with priority 2 and (4, 0); task G with priority 1 and (2, 0). Address questions (a), (b) and (c) from problem 38.

Problem 40 – Consider the same setup as in problem 39 (inherited from problem 38 with different priority assignment for the seven tasks). In this problem we assume task A posts in mailbox B after executing for 1 OS tick and in mailbox C after executing for 2 OS ticks; task B posts in mailbox D after executing for 1 OS tick and in mailbox E after executing for 2 OS ticks; task C posts in mailbox F after executing for 1 OS tick and in mailbox G after executing for 2 OS ticks. Show the schedule for a single-processor system for the first 20 OS ticks. For each OS tick write which task is running (if the processor is idle, write X).