DIGITAL SYSTEMS HT2018

SHEET 4

GABRIEL MOISE

Question 1

In Phos, if the function that forms the body of a process returns, then that function will no longer be on the "ready queue", because it has finished its instructions and does not wait to send or receive any message.

If all the functions that started in the init function return (so they do not loop forever), then the program goes back to the idle process, because that's the only process that remains on the "ready queue" (with priority 3).

Question 2

In Phos, if a process tries to send a message to itself, there are two possibilities:

- the process has a SEND instruction, but not a RECEIVE => this means that the process will be suspended when the SEND instruction is reached and it will wait until the receiver (itself) is ready to receive a message, which won't happen because the process does not have a RECEIVE instruction, so the program will loop forever, or until the queue cannot be stored on the stack anymore
- the process has a SEND instruction that comes before the RECEIVE => when we first get to send the message, the program will be suspended until the receiver is ready and put in the "send queue", but when we go to the receiver (so we begin the process again), we get first to a SEND instruction, so this process is put again in the "sending queue" and so on, until the queue is too big to be handled by the stack

These two cases produce the same outcome, but it felt natural to me to separate them.

Question 3

Let's suppose that there is a cycle in the directed graph, an edge (u,v) meaning that the process u waits to send a message to process v. Now, let's suppose that there is a cycle in the graph $(u_1,u_2,\ldots,u_n,u_1)$, meaning that u_1 is waiting to send to u_2, u_2 is waiting to send to u_3 etc.

For this to happen, the first process u_1 must receive the message from u_n and then u_1 can proceed, but u_n must receive the message from u_{n-1} and then u_n can proceed,…, but u_2 must receive the message from u_1 and then u_2 can proceed, thus each process needs first to receive a message to be able to run and because it is a sending cycle where each process waits for a receive, no progress can be made and this program will loop forever.

We can use the DFS algorithm on the directed graph and if there is a "back" edge in the graph, then there must be a cycle, otherwise there must not be any cycle in the graph.

We can also detect such a cycle with the following approach: we make each process send a variable A to the receiver and a variable B to the receiver of the receiver, this thing happening each time we get into a

process. If, at some point, there is a moment when a process receives both the variable A and the variable B, then we know that we have a cycle, otherwise this approach will terminate when we get to the end of the “send path” that we are following in the graph.

Question 4

In the first situation, when we have the order of the start instructions in init to be:

```
start(USER+0, "Proc1", proc1, 0, STACK);
start(USER+1, "Proc2", proc2, 0, STACK);
```

The output we always obtain is:

```
r = 0
r = 4103
r = 10750
r = 18736
r = 25370
r = 33348
r = 39972
r = 47945
r = 54565
r = 62533
```

The idea behind this is that the scheduling done by the processor, in Phos, is non-preemptive, meaning that the choice of the next process to run is independent of the ones before and a process is suspended only when it enters a sending/receiving state or when an interrupt occurs. After that, the processor picks another process to run. So, as we start with proc1, we get to the part where we call serial_printf, which calls serial_putc, which enters in a sending state to the process called SERIAL. After this, the process which follows is proc2, which increments the **global** value of r until SERIAL is ready to receive and then an interrupt from UART occurs, so we are coming back to proc1, we call serial_printf to r again, which has increased accordingly to the time needed for the SERIAL process to complete and so on…

If the calls to start in init are re-ordered:

```
start(USER+1, "Proc2", proc2, 0, STACK);
start(USER+0, "Proc1", proc1, 0, STACK);
```

Then the output will be:

```
r = 1000000
r = 1000000
r = 1000000
r = 1000000
r = 1000000
```

```
r = 1000000
r = 1000000
r = 1000000
r = 1000000
r = 1000000
```

This can be explained by the fact that proc2 will not be suspended as it does not enter any sending/receiving state throughout the iterations and there is no interrupt from the UART. Thus, r is first increased to 1,000,000 and then, when we go to proc1, this value is printed 10 times, as we have seen.

Question 5

(a) For each character sent, the number of context switches is 2, as we need to produce it, to context switch to the function that sends it to the UART, and then switch back to the function that produces the characters to do this again.

(b) The speed of the UART is 1041 characters/second (9600 bauds), so the time spent on context switches is 2*1041*20µs = 0.04164 seconds, which is 4,164% time spent on the context switches. If we increase the speed of the UART ten times, we get to 10410 characters/second, therefore each second we need 20820 context switches, which means that a total of 0.4164 seconds is allocated for context switches, which is 41,64% and this is, in my opinion, a sensible fraction of the total time needed for the UART to send a character.

(c) A way of reducing the number of context switches per character output would be to add the characters produced (faster than they are printed) into a buffer and then when we get to the printing process, this content of the buffer will be printed and after that we can go back to the process that produces characters and do the same thing again. This means that the number of context switches is reduced significantly as we do not need to come back to the producing process after every character, but only after a period of time.

Question 6

The output of the program showed in Figure 2 is garbled because every time we get to the serial_putc instruction from the put_string procedure, the processor context switches from one process to the other, and because both processes have the same function that they call – “speaker”, but with different arguments, after a character from the first string is printed, a character from the other string follows.

This also justifies why the characters alternate: one from the first slogan, then one from the second one and so on:”nBoR EdXeIaTl MiEsA NbSe … “.

My first solution to this problem involves two speakers that send their messages in turn: “May_task” and “Farage_task” and an intermediate process “presenter” that receives the messages from the 2 processes in turn and then sends them to the put_string procedure. This way, when there is a context switch from the serial_putc to another process, that process can only proceed until it enters the SENDING state and then it has to wait the “presenter” process to be ready to receive, which happens only after the previous slogan has been entirely (and without any sort of garbling) been printed:

```
#include "phos.h"
#include "lib.h"
```

```
#include <string.h>
// Random values chose for the encodings of the names
#define MAY 10
#define FARAGE 11

void put_string(char *s) {
    for (char *p = s; *p != '\0'; p++)
        serial_putc(*p);
}

static const char *slogan[] = {
    "no deal is better than a bad deal\n",
    "BREXIT MEANS BREXIT!\n"
};

void May_task (int n) {
  message m;
  while (1)
  {
    m.m_type = MAY;
    m.m_p1 = slogan[n]; //We embed the slogan in the message to be sent to “presenter”
    send(USER+2, &m);
  }
}

void Farage_task (int n) {
  message m;
  while (1)
  {
    m.m_type = FARAGE;
    m.m_p1 = slogan[n];
    send(USER+2, &m);
  }
}
```

```
void presenter(int n) {
  message m;
  while (1)
  {
    receive(ANY, &m); // We receive any kind of messages
    assert((m.m_sender == USER+0) || (m.m_sender == USER+1)); // The only messages we
need
    put_string(m.m_p1); // Send the slogan to the printing procedure
  }
}


void init(void) {
    serial_init();
    start(USER+0, "May", May_task, 0, STACK);
    start(USER+1, "Farage", Farage_task, 1, STACK);
    start(USER+2, "Presenter", presenter, 0, STACK);
}
```

However, this solution involves passing the message to a “presenter” process, so we will try to avoid that with this second solution:

```
#include "phos.h"
#include "lib.h"
#include <string.h>

#define MAY 10
#define FARAGE 11

static const char *slogan[] = {
    "no deal is better than a bad deal\n",
    "BREXIT MEANS BREXIT!\n"
};

void put_string(char *s) {
  for (char *p = s; *p != '\0'; p++)
        serial_putc(*p);
```

```
}

void speaker (int n){
  message m;

  while (1)
  {
    // Farage waits for May to finish her message
    if (n == 1) receive (ANY, &m);
    put_string(slogan[n]);
    int n1 = 1-n;
    // The message is over
    send(USER+n1, &m);
    // May waits for Farage to finish his message
    if (n == 0) receive (ANY, &m);
  }
}

void init(void) {
    serial_init();
    start(USER+0, "May", speaker, 0, STACK);
    start(USER+1, "Farage", speaker, 1, STACK);
}
```

In this version, both processes have the same function “speaker”, which lets only a slogan to be printed at a time, because in order for a message to be printed it needs to wait until the previous one has been successfully printed and because we start with the process USER+0, the output begins with the message “`no deal is better than a bad deal`” and then we can go on alternating the slogans.

Question 7

The version where we use the sendrec instruction is different from the one where we use both the send and receive as when the process 1 uses sendrec, it enters a SENDING state and when the second process becomes RECEIVING we exchange the message and after that the state of the first process is set immediately to RECEIVING, and we continue with process 2. If we use the send and then receive approach in process 1, first process 1 is in the SENDING state, then when the second process enters a RECEIVING state we can exchange the message, but the state of the first process will after that become READY, although its next instruction is RECEIVE so its state should be RECEIVING. The problem is that we are now in the second process, which became READY and we need to change the state of the first process by using a context switch, thus the sendrec method is more time efficient.