Operating System Testing

Jeremy Williams
3220535

Shawn Downey
3267573

March 11, 2010

Contents

1 Test Cases

1.1 device_test()

This function tests the scheduling for device processes, as well as OS_GetParam(). OS_Yield() is also tested for device processes.

This test simply prints a character to the screen once per scheduling of the device process. The character to be printed is passed in through OS_GetParam().

The desired results are to see a new character on screen only at the rate designated when the process is created.

```
void device_test()
{
    char param[] = {'\0','\0'};
    param[0] = OS_GetParam();
    while(1)
    {
        serial_print(param);
        OS_Yield();
    }
}
```

1.2 scheduling_test()

This function tests a number of things:

1. The proper scheduling of processes according to the defined PPP array.
2. That process creation is limited to MAXPROCESS processes.
3. That two periodics cannot be created at the same time with the same name.
4. That a process that reaches the end of its execution will gracefully terminate automatically.
5. That when there are MAXPROCESS processes and a process terminates, the struct is properly released and another process can be created
6. OS_Yield()

This test starts out by creating three periodic processes (to match the already set PPP array), and also attempts to create two periodic processes with the same name. Next, it proceeds to create as many processes as MAXPROCESS will allow. At this point, the test process should terminate automatically and the processes it created should be turned over to the scheduler.

The first sporadic process to be scheduled attempts to create another new process (since the termination of the test process should have freed room for one more). From this point on each process simply prints a letter indicating itself and then yields.

The desired result is that we see a string of characters representing the processes running in the appropriate order defined by the PPP array.

```
void scheduling_test()
{
    int count = 1;
    OS_Create(&per, 'A', PERIODIC, 'A');
    if(OS_Create(&per, 'A', PERIODIC, 'A') != INVALIDPID)
    {
        serial_print("\nFail: Double periodic created\n");
        count++;
    }
    OS_Create(&per, 'B', PERIODIC, 'B');
    OS_Create(&per, 'C', PERIODIC, 'C');

    count += 2;
    int spor_count = 1;
    while(OS_Create(&spor, spor_count, SPORADIC, 0) != INVALIDPID)
    {
        spor_count++;
        count++;
    }

    if(count != MAXPROCESS-1) //idle proc
        serial_print("\nFail: count != MAXPROCESS\n");

    //OS_Terminate();
}

void per()
{
    char param[] = "\0\0";
    param[0] = OS_GetParam();
    while(1)
    {
        serial_print(param);
        OS_Yield();
    }
}

void spor()
{
    int param = OS_GetParam();
    const char* string;
    if(param == 1)
    {
        if(OS_Create(&spor, 99, SPORADIC, 0) == INVALIDPID)
            serial_print("\nFail: Sporadic 99\n");
    }
    while(1)
    {
        switch(param)
        {
        case 1:
            string = "S1";
            break;
        case 2:
            string = "S2";
            break;
        case 3:
            string = "S3";
            break;
        case 4:
            string = "S4";
            break;
        case 5:
            string = "S5";
            break;
        case 6:
            string = "S6";
            break;
        case 7:
            string = "S7";
            break;
        case 8:
            string = "S8";
            break;
        case 9:
            string = "S9";
            break;
        case 10:
            string = "S10";
            break;
        case 11:
            string = "S11";
            break;
        case 12:
            string = "S12";
            break;
        case 13:
            string = "S13";
            break;
        case 14:
            string = "S14";
            break;
        case 15:
            string = "S15";
            break;
        case 16:
            string = "S16";
            break;
        case 99:
            string = "S99";
            break;
        }

        serial_print(string);
        OS_Yield();
    }
}
```

1.3 sem_test()

This function tests the semaphore related functionality as well as reading and writing to FIFOs.

This test creates an instance of a semaphore and a FIFO, and then creates three "producer" processes. Each of these producers attempts to write four identical characters ('A', 'B', or 'C') to the FIFO, one at a time, through the semaphore. The main test process reads one character at a time from the FIFO and prints it to the screen.

The desired result here is that the output to the output to the screen is 'AAAABBBBCCCC...'.

```
void sem_test()
{
    OS_InitSem(1, 1);
    OS_InitFiFo(1);

    OS_Create(&prod, 1, SPORADIC, 0);
    OS_Create(&prod, 2, SPORADIC, 0);
    OS_Create(&prod, 3, SPORADIC, 0);

    OS_Yield();

    int c;
    char val[] = "\0\0";
    while(1)
    {
        while(!OS_Read(1, &c))
            OS_Yield();
        val[0] = c;
        serial_print(val);
        OS_Yield();
    }
}

void prod()
{
    int param = OS_GetParam();
    char c;

    switch(param)
    {
    case 1:
        c = 'A';
        break;
    case 2:
        c = 'B';
        break;
    case 3:
        c = 'C';
        break;
    }

    int count = 0;
    while(1)
    {
        OS_Wait(1);
        OS_Write(1, c);
        if(count == 3)
        {
            count = -1;
            OS_Signal(1);
        }
        count++;
        OS_Yield();
    }
}
```

2 Failures

2.1 device_test()

Device processes appear to be running faster than their designated rate.

Note: It is possible that the simulator is running at a faster rate than the robot would be, in which case this failure may not be an issue.

2.2 scheduling_test()

1. Processes do not appear to terminate gracefully without an explicit call to OS_Terminate()
2. When periodic processes yield before their time quantum is up, instead of scheduling a sporadic, the next periodic is scheduled
3. As a consequence of the above issue, sporadic processes are only scheduled during time defined as IDLE in the PPP array (as opposed to during the IDLE slot as well as during periodic "down-time")

2.3 sem_test()

1. First process to call OS_Wait does not execute its critical section right away
2. Calling OS_Wait when you already have the semaphore acts strangely
3. OS_Signal does not appear to release the semaphore
4. Processes waiting on a semaphore get scheduled regardless (possibly improperly/non implemented wait queue)