Operating systems and concurrency B05

David Kendall

Northumbria University

Introduction

- Multi-tasking program from previous lecture is very simple:
  - No need for communication between tasks
  - No shared resources
  - No need for synchronisation
- Most multi-tasking programs are not so simple:
  - Communication: shared variables; message-passing
  - Shared resources: interference or race conditions
  - Synchronisation: critical sections; mutual exclusion

Multi-tasking program with sharing

- Let's look at a slightly more (artificially) complicated version of the example from last week (main.c)
- Notice there is a boolean variable `flashing` that is initially false and must become true in order for the lights to start flashing
- There are 3 shared `uint32_t` variables: `total`, `count1` and `count2`
- There are 2 new tasks: `appTaskCount1` and `appTaskCount2`
- The tasks increment their `count` variables and the `total` and check that `count1 + count2` is equal to `total`: if not start flashing.

`appTaskCount1` behaviour

```
static void appTaskCount1(void *pdata) {
  while (true) {
    count1 += 1;
    display(1, count1);
    total += 1;
    if ((count1 + count2) != total) {
      flashing = true;
    }
    OSTimeDlyHMSM(0,0,0,20);
  }
}
```

- `appTaskCount2` is similar: it increments and displays `count2` (not `count1`)

QUESTION

QUESTION

Will the lights start flashing?

Working towards an answer

- Look at the crucial parts of `appTaskCount1` and `appTaskCount2`

appTaskCount1	appTaskCount2
A.1 `count1 += 1;`	B.1 `count2 += 1;`
A.2 `display(1, count1);`	B.2 `display(2, count2);`
A.3 `total += 1;`	B.3 `total += 1;`
A.4 `if ...`	B.4 `if ...`

- What is the value of `total` at A.4 and B.4 in each case below (assume all values initially 0):
  - A.1, A.2, A.3, A4, B.1, B.2, B.3, B4

Working towards an answer

- Look at the crucial parts of `appTaskCount1` and `appTaskCount2`

appTaskCount1	appTaskCount2
A.1 count1 += 1;	B.1 count2 += 1;
A.2 display(1, count1);	B.2 display(2, count2);
A.3 total += 1;	B.3 total += 1;
A.4 **if** ...	B.4 **if** ...

- What is the value of `total` at A.4 and B.4 in each case below (assume all values initially 0):
  - A.1, A.2, A.3, A4, B.1, B.2, B.3, B4
  - B.1, B.2, A.1, A.2, A3, A4, B3, B4

Question and Answer

- Question: Will the lights start flashing?

Question and Answer

- Question: Will the lights start flashing?
- Answer: MAYBE

Question and Answer

- Question: Will the lights start flashing?
- Answer: MAYBE
- It depends on the scheduler and when tasks become ready to run.

Question and Answer

- Question: Will the lights start flashing?
- Answer: MAYBE
- It depends on the scheduler and when tasks become ready to run.
- Can `appTaskCount2` ever interfere with `appTaskCount1`?

Question and Answer

- Question: Will the lights start flashing?
- Answer: MAYBE
- It depends on the scheduler and when tasks become ready to run.
- Can `appTaskCount2` ever interfere with `appTaskCount1`?
  - No - it's a lower priority task; `appTaskCount1` never blocks in its critical section, so `appTaskCount2` never has a chance to interfere with it.

Interference - summary

- What is the problem?
  - Interference
  - One or more tasks is prevented from generating a correct result because of interference from another task
  - Sometimes known as a race condition
- Why is it caused?
  - Arbitrary interleaving of task instructions
  - created by the scheduler
- How can it be prevented?
  - Avoid shared variables, or
  - Enforce mutual exclusion of critical sections

How to enforce mutual exclusion of critical sections

- Memory interlock
- Mutual exclusion algorithms: Dekker, Peterson, Lamport
- Disable interrupts
  - `OS_ENTER_CRITICAL(), OS_EXIT_CRITICAL()`
  - Use with extreme caution – preferably not at all.
- Semaphores
- Monitors

Mutual exclusion of critical sections

- A critical section is part of a program in which a shared resource is accessed: global variable, file, etc.
- Mutual exclusion is the requirement that no more than one process is executing its critical section at the same time
- An acceptable solution to the mutual exclusion problem requires several properties:
  1. Mutual exclusion is enforced
  2. No deadlock
  3. No livelock (starvation)
  4. No requirement for strict alternation (if other process doesn't need access to c.s. then a process should be able to enter its c.s. immediately)

Peterson's algorithm for mutual exclusion

- Difficult to get a correct solution to mutual exclusion problem
- Many incorrect attempts
  - Perhaps instructive to look at some of them – later.
- Peterson proposed a correct algorithm (main.c)

Careful look at Peterson's algorithm

```
static void appTaskCount1(void *pdata) {
  while (true) {

    need1 = true;
    turn = 2;
    while (need2 && (turn == 2)) {
      // OSTimeDlyHMSM(0,0,0,1);
    }

    count1 += 1;
    display(1, count1);
    total += 1;

    need1 = false;

    if ((count1 + count2) != total) {
      flashing = true;
    }
    OSTimeDlyHMSM(0,0,0,20);
  }
```

Careful look at Peterson's algorithm

```
static void appTaskCount1(void *pdata) {
  while (true) {

    need1 = true;
    turn = 2;
    while (need2 && (turn == 2)) {
      // OSTimeDlyHMSM(0,0,0,1);
    }

    count1 += 1;
    display(1, count1);
    total += 1;

    need1 = false;

    if ((count1 + count2) != total) {
      flashing = true;
    }
    OSTimeDlyHMSM(0,0,0,20);
  }
```

ENTRY PROTOCOL

Careful look at Peterson's algorithm

```
static void appTaskCount1(void *pdata) {
  while (true) {

    need1 = true;
    turn = 2;
    while (need2 && (turn == 2)) {
      // OSTimeDlyHMSM(0,0,0,1);
    }

    count1 += 1;
    display(1, count1);
    total += 1;

    need1 = false;

    if ((count1 + count2) != total) {
      flashing = true;
    }
    OSTimeDlyHMSM(0,0,0,20);
  }
```

ENTRY PROTOCOL

CRITICAL SECTION

Careful look at Peterson's algorithm

```
static void appTaskCount1(void *pdata) {
  while (true) {

    need1 = true;
    turn = 2;
    while (need2 && (turn == 2)) {
      // OSTimeDlyHMSM(0,0,0,1);
    }

    count1 += 1;
    display(1, count1);
    total += 1;

    need1 = false;

    if ((count1 + count2) != total) {
      flashing = true;
    }
    OSTimeDlyHMSM(0,0,0,20);
  }
```

ENTRY PROTOCOL

CRITICAL SECTION

EXIT PROTOCOL

BUSY WAITING