Operating systems and concurrency - B05

David Kendall

Northumbria University

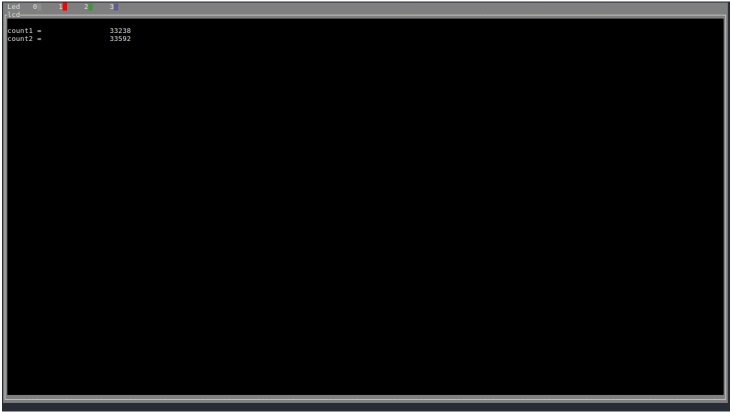
Introduction

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

Multi-threaded program with sharing

- Let’s look at a slightly more (artificially) complicated example
- There is a boolean variable `flashing` that is initially false and must become true in order for a light on a console to start flashing
- There are 3 shared variables: `total`, `count1` and `count2`
- There are 2 new threads: `count1_thr` and `count2_thr`
- The threads increment their `count` variables and the `total` and check that `count1 + count2` is equal to `total`: if not start flashing.
- Once flashing starts, the threads stop counting and just sit in a tight loop.

An example console



```
Led   0     1     2     3
lcd

count1 =                33238
count2 =                33592
```

- The console has WHITE, RED, GREEN and BLUE leds
- It has a display (lcd) for writing text
- In this example the RED light is flashing

count1_thr behaviour

```
void *count1_thr(void * arg) {
    while (!flashing) {
        count1 += 1;
        total += 1;
        if ((count1 + count2) != total) {
            flashing = true;
        }
        lcd_write_at(1, 0, "count1 = %20d", count1);
    }
    while (true) {
        /* skip */
    }
}
```

- count2_thr 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 `count1_thr` and `count2_thr`

```
      count1_thr
A.1  count1 += 1;
A.2  total += 1;
A.3  if ...
```

```
      count2_thr
B.1  count2 += 1;
B.2  total += 1;
B.3  if ...
```

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

Working towards an answer

- Look at the crucial parts of `count1_thr` and `count2_thr`

```
        count1_thr                       count2_thr
A.1  count1 += 1;              B.1  count2 += 1;
A.2  total += 1;               B.2  total += 1;
A.3  if ...                    B.3  if ...
```

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

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 threads become ready to run.

Interference - summary

- What is the problem?
  - Interference
  - One or more threads is prevented from generating a correct result because of interference from another thread
  - Sometimes known as a race condition
- Why is it caused?
  - Arbitrary interleaving of thread 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
- Semaphores
- Monitors
- Look at Peterson's algorithm today — more on the rest later

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 which we look at next.

Careful look at Peterson's algorithm

```
void *count1_thr(void * arg) {
    while (!flashing) {

    need1 = true;
    turn = 2;
    while (need2 && (turn == 2)) {
        /* busy wait */
    }

    count1 += 1;
    total += 1;
    if ((count1 + count2) != total) {
        flashing = true;
    }
    lcd_write_at(1, 0, "count1 = %20d", count1);

    need1 = false;
```

Careful look at Peterson's algorithm

```
void *count1_thr(void * arg) {
    while (!flashing) {

    need1 = true;
    turn = 2;
    while (need2 && (turn == 2)) {
        /* busy wait */
    }

    count1 += 1;
    total += 1;
    if ((count1 + count2) != total) {
        flashing = true;
    }
    lcd_write_at(1, 0, "count1 = %20d", count1);

    need1 = false;
```

ENTRY PROTOCOL

Careful look at Peterson’s algorithm

```
void *count1_thr(void * arg) {
    while (!flashing) {

    need1 = true;
    turn = 2;
    while (need2 && (turn == 2)) {
        /* busy wait */
    }

    count1 += 1;
    total += 1;
    if ((count1 + count2) != total) {
        flashing = true;
    }
    lcd_write_at(1, 0, "count1 = %20d", count1);

    need1 = false;
```

ENTRY PROTOCOL

CRITICAL SECTION

Careful look at Peterson's algorithm

```
void *count1_thr(void * arg) {
    while (!flashing) {

    need1 = true;
    turn = 2;
    while (need2 && (turn == 2)) {
        /* busy wait */
    }
```

ENTRY PROTOCOL

```
    count1 += 1;
    total += 1;
    if ((count1 + count2) != total) {
        flashing = true;
    }
    lcd_write_at(1, 0, "count1 = %20d", count1);
```

CRITICAL SECTION

```
    need1 = false;
```

EXIT PROTOCOL

A problem with Peterson's algorithm

BUSY WAITING