University of Southampton

COMP2212 Programming Language Concepts

Shared Variable Concurrency

Dr Julian Rathke

Processes, address spaces

- A **process** is an Operating System abstraction. Typically a process involves an
  - address space
  - a number of **threads**
    - each with its own call-stack,
    - threads within a single process share an address space and thus can communicate via **shared memory**
    - threads can read and write to memory via a local cache
    - on multi/many core systems threads write to **different** caches!
    - Cache synchronisation is computationally expensive so writes to main memory (heap) may not take effect immediately
    - Threads on different cores will not see each other's writes until the caches are flushed to main memory.
- An operation by one thread is **visible** in another thread if its effect can be seen via main memory.

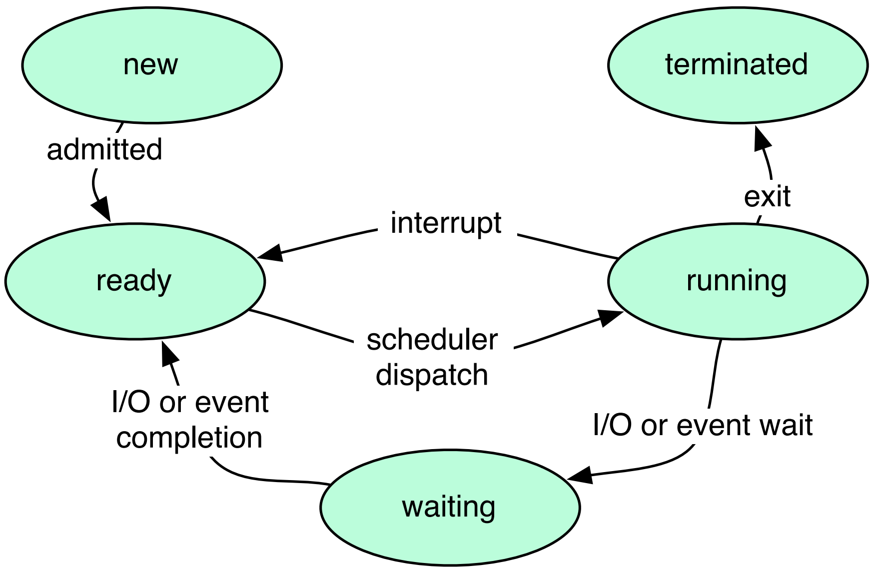
Shared memory concurrency basics

- Fundamentally, for two threads to communicate using shared memory the following happens
  - The sending thread must write a message in to a memory location
  - The sending thread notifies any receiver threads that it has done so
  - The receiving thread reads the message from the same shared memory location
  - For synchronous communication, the receiving thread notifies the sender that it has done so
- There is a order to the above points - without any other control, the thread scheduler chooses whether the sender or receiver will execute next.
  - This can be a problem!

Threads and context switch

- The operation of switching control from one thread to another is called **context switch**
  - context switches happen at the granularity of machine level instructions
  - a context switch can happen in the middle of a "high-level" operation (e.g. assignment to a variable)
  - Compilers **and** processors may reorder certain instructions for efficiency.

Thread life cycle



Conclusion : Shared memory concurrency is **really hard**!

Race conditions

Consider the following C threading code (using pthreads)

```c
#include <stdio.h>
#include <pthread.h>
#define NUM_OF_TRANS 10

double accountBalance;

void withdraw(double outAmount) {
    if (accountBalance > outAmount) {
            accountBalance = accountBalance - outAmount;
            printf("Withdrew £%.2f. Your balance is now £%.2f.\n",
                outAmount, accountBalance);
        }
    else printf("You don't have enough cash!\n");
}

void credit(double inAmount) {
    accountBalance = accountBalance + inAmount;
    printf("Credited £%.2f. Your balance is now £%.2f.\n",
        inAmount, accountBalance);
}
```

Continued Over

Race conditions

```
void *transaction(void *arg)
{
    double amount = *(double *)arg;
    if (amount < 0) withdraw(-amount);
    else if (amount > 0) credit(amount);
    pthread_exit(NULL);
}

int main()
{
    double args[NUM_OF_TRANS] = {-5.0, 10.0, -15.0, 3.0,
        -20.0, -50.0, 10.0, 15.0, 20.0, -10.0};
    pthread_t threads[NUM_OF_TRANS];
    accountBalance = 100.0;

    int i;
    for (i=0;i<NUM_OF_TRANS;i++) {
        (void)pthread_create(&threads[i],NULL,transaction,&args[i]);
    }
    pthread_exit(NULL);
}
```

What happens if we run withdraw(10.0) and credit(20.0) as two threads, supposing that accountBalance is initially 100.0?

Experiments

University of Southampton

test 1

```
Withdrew £5.00. Your balance is now £95.00.
Credited £10.00. Your balance is now £105.00.
Credited £3.00. Your balance is now £108.00.
Withdrew £20.00. Your balance is now £88.00.
Withdrew £50.00. Your balance is now £38.00.
Credited £10.00. Your balance is now £48.00.
Credited £15.00. Your balance is now £63.00.
Credited £20.00. Your balance is now £83.00.
Withdrew £10.00. Your balance is now £73.00.
Withdrew £15.00. Your balance is now £58.00.
~/teaching/COMP2009 >
```

different interleaving

```
Withdrew £5.00. Your balance is now £95.00.
Credited £10.00. Your balance is now £105.00.
Withdrew £15.00. Your balance is now £90.00.
Credited £3.00. Your balance is now £93.00.
Withdrew £20.00. Your balance is now £73.00.
Withdrew £50.00. Your balance is now £23.00.
Credited £10.00. Your balance is now £33.00.
Credited £15.00. Your balance is now £48.00.
Credited £20.00. Your balance is now £68.00.
Withdrew £10.00. Your balance is now £58.00.
~/teaching/COMP2009 >
```

hmm...

```
Withdrew £5.00. Your balance is now £95.00.
Credited £10.00. Your balance is now £105.00.
Withdrew £20.00. Your balance is now £73.00.
Credited £15.00. Your balance is now £48.00.
Withdrew £15.00. Your balance is now £90.00.
Credited £20.00. Your balance is now £68.00.
Withdrew £50.00. Your balance is now £23.00.
Withdrew £10.00. Your balance is now £58.00.
Credited £3.00. Your balance is now £93.00.
Credited £10.00. Your balance is now £33.00.
~/teaching/COMP2009 >
```

what's going on?!?!?!

non-deterministic madness!

```
Withdrew £5.00. Your balance is now £95.00.
Credited £10.00. Your balance is now £105.00.
Credited £3.00. Your balance is now £93.00.
Withdrew £50.00. Your balance is now £43.00.
Credited £10.00. Your balance is now £53.00.
Credited £15.00. Your balance is now £68.00.
Credited £20.00. Your balance is now £88.00.
Withdrew £10.00. Your balance is now £78.00.
Withdrew £20.00. Your balance is now £58.00.
Withdrew £15.00. Your balance is now £90.00.
~/teaching/COMP2009 >
```

```
Withdrew £5.00. Your balance is now £95.00.
Credited £10.00. Your balance is now £105.00.
Withdrew £15.00. Your balance is now £90.00.
Credited £3.00. Your balance is now £93.00.
Credited £10.00. Your balance is now £103.00.
Credited £15.00. Your balance is now £118.00.
Credited £20.00. Your balance is now £68.00.
Withdrew £10.00. Your balance is now £58.00.
Withdrew £20.00. Your balance is now £98.00.
Withdrew £50.00. Your balance is now £48.00.
~/teaching/COMP2009 >
```

```
Withdrew £5.00. Your balance is now £95.00.
Credited £10.00. Your balance is now £105.00.
Credited £3.00. Your balance is now £93.00.
Credited £10.00. Your balance is now £33.00.
Withdrew £50.00. Your balance is now £23.00.
Withdrew £10.00. Your balance is now £58.00.
Withdrew £20.00. Your balance is now £73.00.
Credited £15.00. Your balance is now £48.00.
Withdrew £15.00. Your balance is now £90.00.
Credited £20.00. Your balance is now £68.00.
~/teaching/COMP2009 >
```

```
Withdrew £5.00. Your balance is now £95.00.
Credited £3.00. Your balance is now £98.00.
Credited £15.00. Your balance is now £48.00.
Credited £20.00. Your balance is now £68.00.
Withdrew £10.00. Your balance is now £58.00.
Withdrew £50.00. Your balance is now £13.00.
Withdrew £15.00. Your balance is now £83.00.
Withdrew £20.00. Your balance is now £63.00.
Credited £10.00. Your balance is now £33.00.
Credited £10.00. Your balance is now £23.00.
~/teaching/COMP2009 >
```

Causes of race conditions

- A race condition describes the situation in which two or more threads are simultaneously trying to write and read-or-write from shared memory.
  - This can't actually happen simultaneously so one thread will end up writing or reading before the others.
  - In the absence of any other control, it is the thread scheduler that decides which.
- Race conditions become possible when both of the following are present
  - **Aliasing** - same location in shared memory (heap) accessible from multiple threads
  - **Mutability** - data on the heap can be altered
- This suggests that to avoid race conditions we can limit one of aliasing or mutability to single threads.
  - This is the approach taken in languages such as Rust and Clojure
- Another approach is to carefully program to avoid or control areas of code where shared memory is written to.

Critical regions and Mutual Exclusion

University of Southampton

- Let's start with a couple of common concepts:
- **critical region**: part of program where a shared resource is accessed
- **mutual exclusion:** if one thread is in its critical region for a resource then no other threads are allowed to enter their critical regions for that resource.
  - if no two threads are in a critical region at the same time then there will be no races!
- So how do we ensure mutual exclusion in a critical section?
- The naive approach is to simply use a boolean flag acting as a lock. Before entering the critical section a thread must check the flag and only enter if no other thread has already set the lock.

Solution 1 - Single lock

Thread 0

```
while(1) {
    while (lock);
    lock = 1;
    critical_region();
    lock = 0;
    noncritical_region();
}
```

Thread 1

```
while(1) {
    while (lock);
    lock = 1;
    critical_region();
    lock = 0;
    noncritical_region();
}
```

- lock has initial value 0
- Does this guarantee mutual exclusion?
- No, because there may be a context switch after both threads have checked the lock in the while (lock) statement
    - Both threads would then think the lock is available and both would enter the critical region.

Solution 2 - Taking turns

Thread 0

```
while(1) {
    while (turn != 0);
    critical_region();
    turn = 1;
    noncritical_region();
}
```

Thread 1

```
while(1) {
    while (turn != 1);
    critical_region();
    turn = 0;
    noncritical_region();
}
```

- This alternative "solution" has the threads alternating.
- What properties does this implementation satisfy?
- It does guarantee mutual exclusion.
- But is it a satisfactory solution?
- Not really, as it doesn't support truly concurrent behaviour.
    - Also, the system may get stuck if it is Thread1's turn but Thread1 isn't ready to enter this code.

Peterson's Algorithm

called before entering critical region

called after exiting critical region

```
#define FALSE   0
#define TRUE    1
#define N       2     // number of processes

int turn;
int interested[N];

void enter_region(int process) {
    int other;
    other = 1-process;
    interested[process] = TRUE;
    turn = process;
    while ( turn == process && interested[other] == TRUE );
}

void leave_region(int process) {
    interested[process] = FALSE;
}
```

This algorithm mixes the idea of boolean locks and turn taking to guarantee mutual exclusions, blocked threads and avoids unbounded waiting for other threads.

It uses the technique of **busy waiting** to make threads wait and to keep checking the locks and turn variables.

Reordering

University of Southampton

- Modern compilers & hardware **reorder instructions** within individual threads
    - this is done consistently so that each individual CPU still follows the logic of the original
    - however, the order in which one thread sees changes in memory made by another thread on a different CPU may not follow the logic of the source code
- this can break down classical algorithms such as Peterson's
    - Consider what happens if one of the threads instead appears to execute

```
...
turn = process
interested[process] = TRUE
...
```

a context switch right here could break mutual exclusion

```
T0                  |   T1
...                 | turn = 1
int[0] = TRUE       | ...
turn = 0            | ...
while (check)       | ...
enters CS           | ...
...                 | int[1] = TRUE
...                 | while (check)
...                 | enters CS
...                 | ...
```

Memory Barriers

University of Southampton

- Memory barriers (or fences) are machine level instructions that are used to help with problems due to reordering
- They preserve the externally visible program order between CPUs
- They also make memory visible by causing cache propagation.
- Examples include
    - sfence, ifence and mfence on x86 architecture
    - Instruction Synchronisation Barrier (ISB) on Arm architecture
- Inserting a memory barrier before the `while` check in Peterson's algorithm would guarantee mutual exclusion.

Modern Hardware support

- Home-cooked solutions are non-trivial and sometimes break with modern hardware
  - memory visibility, compiler optimisations, hardware pipelines, etc.
- Modern architectures now provide additional support for locking in the form of atomic operations.
- Atomic operations provided by hardware allow for testing and setting of a lock as a single, uninterruptible operation.
- For example:
  - Test-and-set lock (TSL) instruction TSL RX,LOCK
    - reads the value of memory location LOCK
    - writes the value to register RX
    - stores a nonzero value at LOCK
  - guaranteed to be indivisible: memory bus is locked for the duration of operation
    - other CPUs cannot interfere
- Such support allows us to provide mutual exclusion for critical sections much more easily.

Mutual exclusion with TSL

- Atomicity (hardware-guaranteed) of TSL ensures that races are avoided
- In programming languages a similar operation is sometimes given as a language primitive called CAS (compare-and-set)
  - CAS is used for programming **fine-grained concurrent algorithms**

```
enter_region:
    TSL RX,LOCK
    CMP RX,#0            | check LOCK was 0
    JNE enter_region     | not 0 then try again
    RET                  | 0 so clear to enter critical region

leave_region:
    MOV LOCK,#0
    RET
```

Mutual exclusion with XCHG

- XCHG exchanges two register and memory location atomically
  - used for similar purposes as TSL
  - atomicity of operation is the crucial ingredient
- used in Intel x86 CPUs

```
enter_region:
    MOV RX,#1           | set RX to 1
    XCHG RX,LOCK        | atomically exchange values of RX and LOCK
    CMP RX,#0
    JNE enter_region
    RET

leave_region:
    MOV LOCK,#0
    RET
```

Next Lecture

Locks and Synchronisation