COMP2212 Programming Language Concepts

Threads, races and hardware
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 and context switch

- The operation of switching control from one thread to another is called **context switch**
  - context switch happens 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)
- On multi/many core, threads can run truly concurrently on different cores (and so have different caches! we will talk about **visibility** later this week)
- Shared memory concurrency is hard!
What happens if we run withdraw(10.0) and credit(20.0) as two threads, supposing that accountBalance is initially 100.0?
Threads in C

PTHREAD_CREATE(3)  BSD Library Functions Manual  PTHREAD_CREATE(3)

NAME
pthread_create -- create a new thread

SYNOPSIS
#include <pthread.h>

int
pthread_create(pthread_t *restrict thread, const pthread_attr_t *restrict attr, void *(*start_routine)(void *),
               void *restrict arg);

DESCRIPTION
The pthread_create() function is used to create a new thread, with attributes specified by attr, within a process. If
attr is NULL, the default attributes are used. If the attributes specified by attr are modified later, the thread's
attributes are not affected. Upon successful completion, pthread_create() will store the ID of the created thread in the
location specified by thread.

Upon its creation, the thread executes start_routine, with arg as its sole argument. If start_routine returns, the
effect is as if there was an implicit call to pthread_exit(3), using the return value of start_routine as the exit status.
Note that the thread in which main() was originally invoked differs from this. When it returns from main(), the effect
is as if there was an implicit call to exit(3), using the return value of main() as the exit status.

The signal state of the new thread is initialized as:

- The signal mask is inherited from the creating thread.
- The set of signals pending for the new thread is empty.

RETURN VALUES
If successful, the pthread_create() function will return zero. Otherwise, an error number will be returned to indicate the
error.

ERRORS
pthread_create() will fail if:

[EAGAIN] The system lacked the necessary resources to create another thread, or the system-imposed limit on the
total number of threads in a process [PTHREAD_THREADS_MAX] would be exceeded.
PThreads implementation

```c
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);
}
```
Experiments

Test 1

Different interleaving

HMM...

What's going on?!

Non-deterministic madness!
Critical regions

- one solution to problems with shared resources: memory, I/O, files etc
- **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
Solution 1

Thread 1

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

Thread 2

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

- `lock` has initial value 0
- Does this guarantee mutual exclusion?
Solution 2

Thread 1

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

Thread 2

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

• What properties does this implementation satisfy?
• Is it a satisfactory solution?
Peterson’s Algorithm

- Interesting for historical reasons

```c
#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;
}
```
Busy waiting

• The various implementations we have discussed employ **busy waiting**
  • blocked threads keep checking the status of lock & turn variables
• Modern compilers & hardware **reorder instructions** within individual threads
  • (weak) memory models
    • more about this later (Java memory model)
• this can break down classical algorithms such as Peterson’s
Hardware support

- Home-cooked solutions are non-trivial and sometimes break with modern hardware
  - memory visibility, compiler optimisations, hardware pipelines, etc.
- Atomic operations provided by hardware
  - Test-and-set lock (TSL) instruction \texttt{TSL RX, LOCK}
    - reads the value of memory location \texttt{LOCK}
    - writes the value to register \texttt{RX}
    - stores a nonzero value at \texttt{LOCK}
    - guaranteed to be indivisible: memory bus is locked for the duration of operation
  - other CPUs cannot interfere
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
  - more on this next week
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
  XCHG RX,LOCK
  CMP RX,#0
  JNE enter_region
  RET

leave_region:
  MOV LOCK,#0
  RET
```