G52OSC
OPERATING SYSTEMS AND CONCURRENCY

Atomic Operations

Dr Jason Atkin

Last lecture

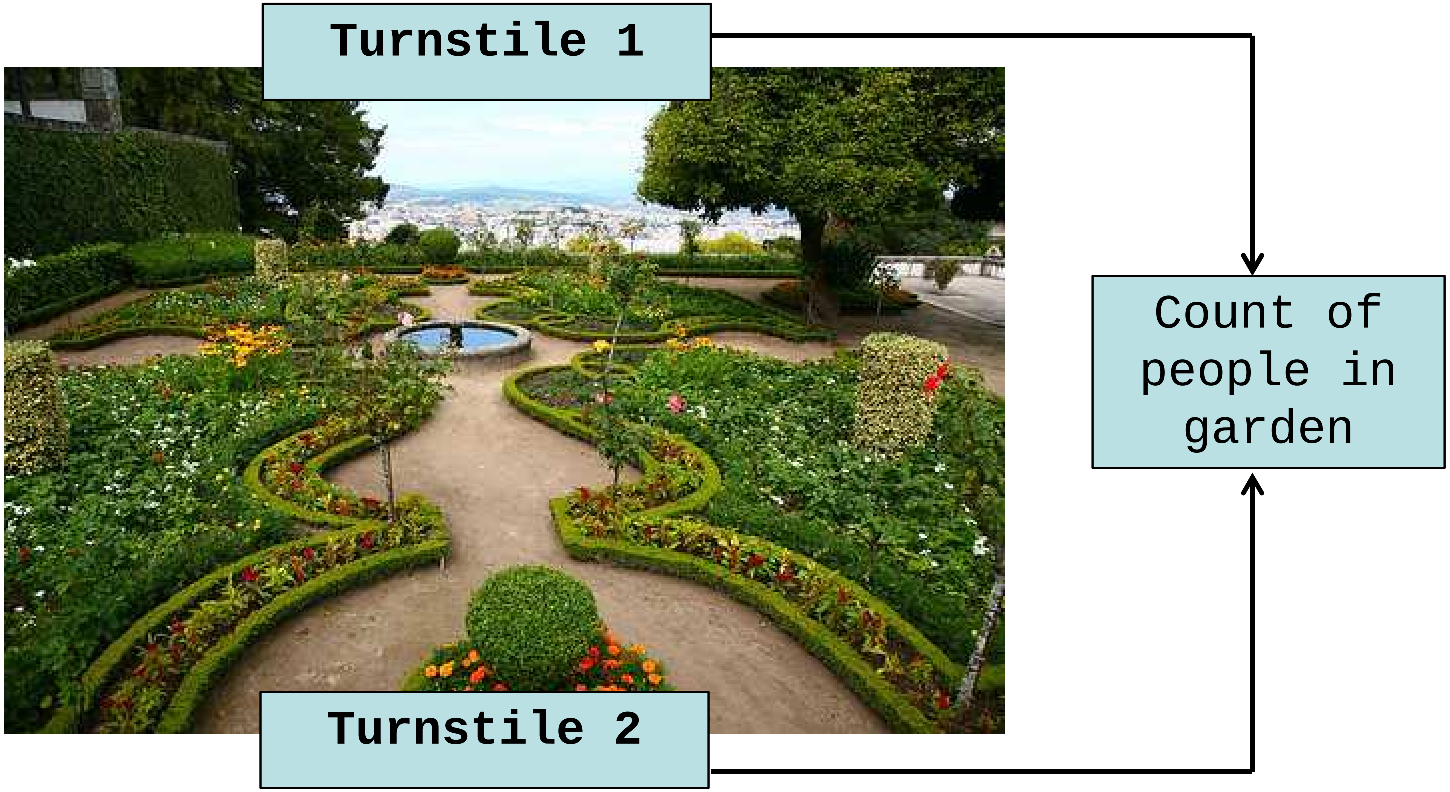
- Sharing memory
- Synchronising processes/threads using memory
  - Examining a variable set by another process
- Threads share an address space
- Processes have their own address space
  - But you can share memory between them
    - Have to do it manually!
  - Using a struct pointer is a simple way to manage this shared memory

This lecture

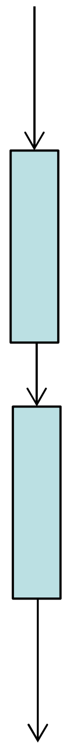
- Coordinating access to data
- Ornamental garden problem again
  - Why it went wrong
  - How it can be fixed
- Thread traces
- Atomic operations
- Critical sections
- Spin-locks

An example concurrency problem

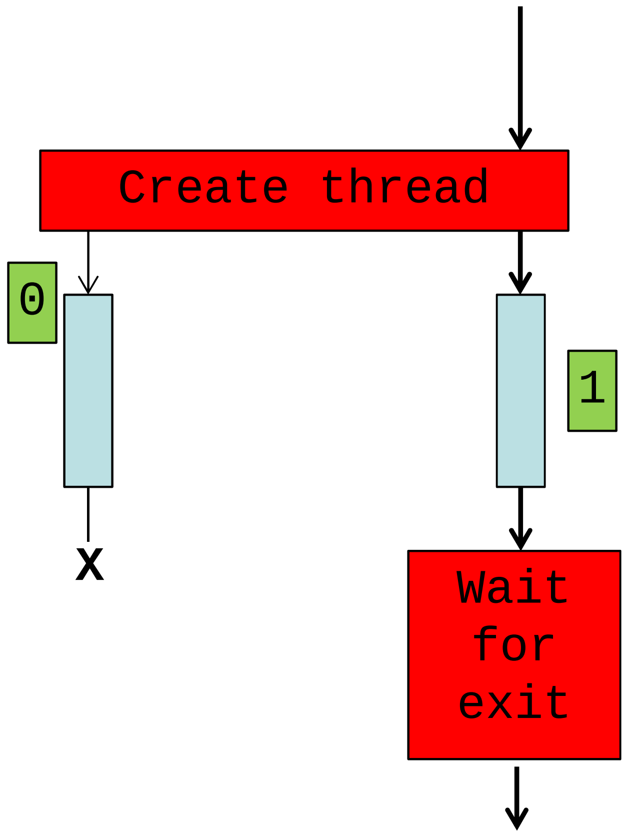
- Ornamental garden problem:
  - Search for: "ornamental garden" concurrency
- Each should increment a **shared** counter when someone enters the garden



Two-thread update



- We create an extra thread
- We run the function twice
  - One from each thread
  - Including initial one
- Each function increments the value 1 million times
- What is the final value?



Example function

```
#include <Windows.h>
#include <stdio.h>
#define NUM_THREADS 2

// Windows name for an unsigned long - 4 bytes
volatile DWORD dwTotal = 0;

// Function called by every thread
DWORD WINAPI thread_function( LPVOID lParam )
{
      for ( int i = 0; i < 1000000; i++ )
      {
            dwTotal++;
      }
      return 0;
}
```

Contents of main()

```
HANDLE arrdwThreadHandles[NUM_THREADS];

for ( int iTN = 0; iTN < NUM_THREADS - 1; ++iTN )
    arrdwThreadHandles[iTN] = CreateThread(
            NULL, 0,
            thread_function, /* Name of function to call */
            (LPVOID)iTN, /*parameter you can give*/
            0, NULL
        );

/* Do the last one in the current thread */
thread_function( (LPVOID)(NUM_THREADS - 1) );

WaitForMultipleObjects( NUM_THREADS - 1,
    arrdwThreadHandles, /* Array of handles */
    TRUE, 10000 ); /* Wait for all, for up to 10 secs */
```

What is the result when run twice?

```
// Function called by every thread
DWORD WINAPI thread_function(LPVOID lParam)
{
    for ( int i = 0; i < 1000000; i++ )
    {
        dwTotal++;
    }
    return 0;
}
```

Each function increments the variable one million times

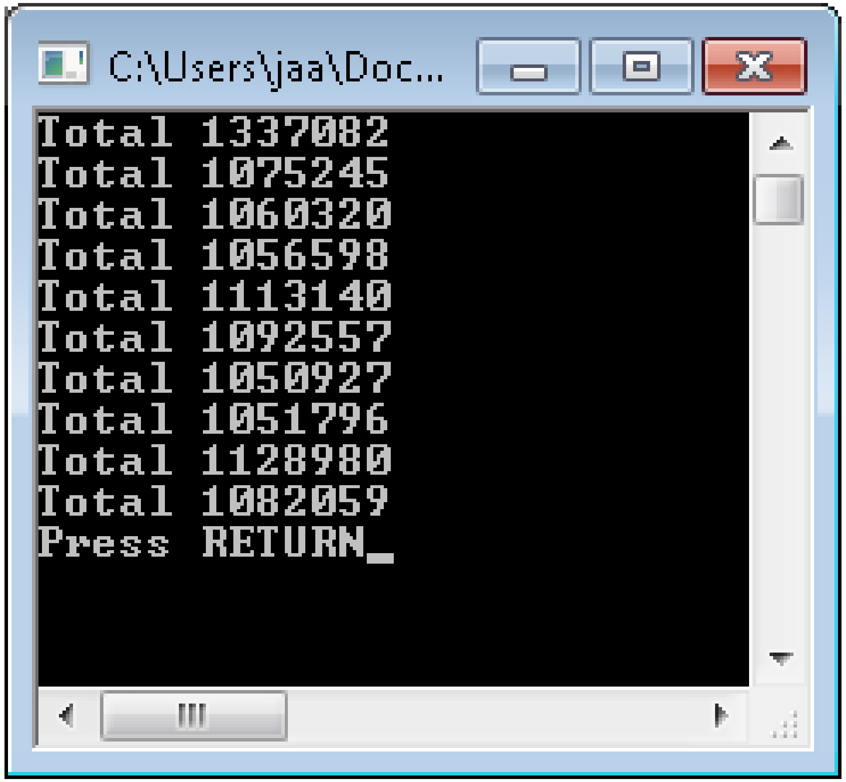
Run the function twice, at the same time

What will the result be?

Full code (for copy-paste)

```
int main()
{
   HANDLE arrdwThreadHandles[NUM_THREADS];

   for ( int iTN = 0; iTN < NUM_THREADS - 1; ++iTN )
      arrdwThreadHandles[iTN] = CreateThread( NULL,0,
         thread_function, /* Name of function to call */
   (LPVOID)iTN, /*parameter you can give*/ 0,NULL );

   /* Do the last one in the current thread */
   thread_function( (LPVOID)(NUM_THREADS - 1) );

   WaitForMultipleObjects( NUM_THREADS - 1,
         arrdwThreadHandles, /* Array of handles */
         TRUE,10000 ); /* Wait for all, for up to 10 secs */

   printf( "%d\n",dwTotal );
   while ( getchar() != '\n' );
}
```

Results



- Should have been 2,000,000
- Problem:

  `dwTotal++;`

  – Get current value

  – Increment

  – Write new value back

What went wrong?

- What does the increment do?

1. Load current value from memory into a register
2. Increment the register
3. Write value back again to memory

- What could go wrong?

What should have happened?

Thread 1	T1		Mem		T2	Thread 2
Get value	0	←	0			
Increment value	1		0			
Write value	1	→	1			
			1	→	1	Get value
			1		2	Increment value
			2	←	2	Write value
Get value	2	←	2			
Increment value	3		2			
Write value	3	→	3			

Example of what can go wrong

Thread 1	T1		Mem		T2	Thread 2
Get value	**0**	←	0			
	0		0	→	**0**	Get value
Increment value	**1**		0		0	
	1		0		**1**	Increment value
Write value	1	→	**1**		1	
			1	←	1	Write value
Get value	**1**	←	1			
	1		1	→	**1**	Get value
Increment value	**2**		1		1	

Traces

- Thread 1

- Get value
- Increment value
- Write value

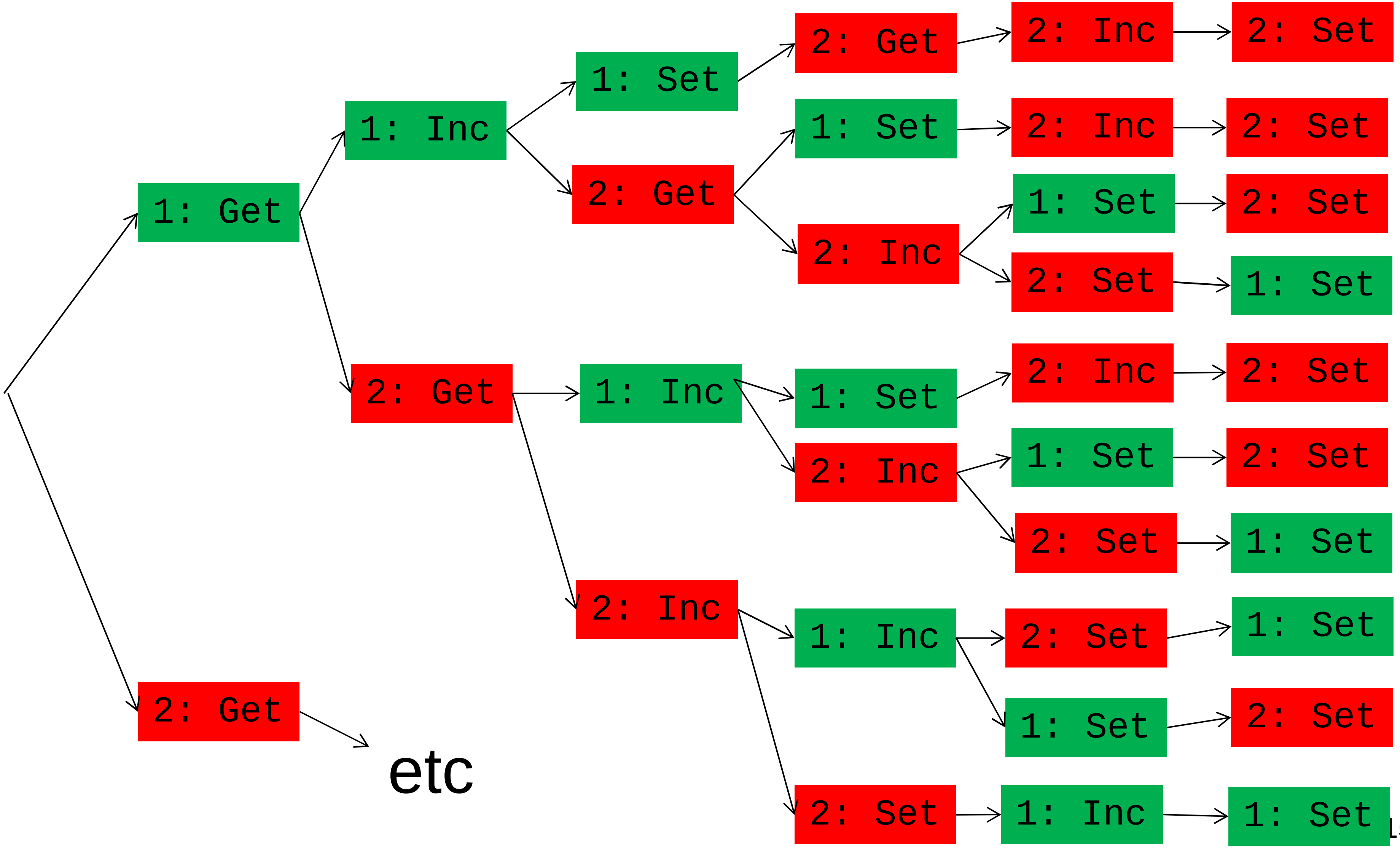
- Thread 2

- Get value
- Increment value
- Write value

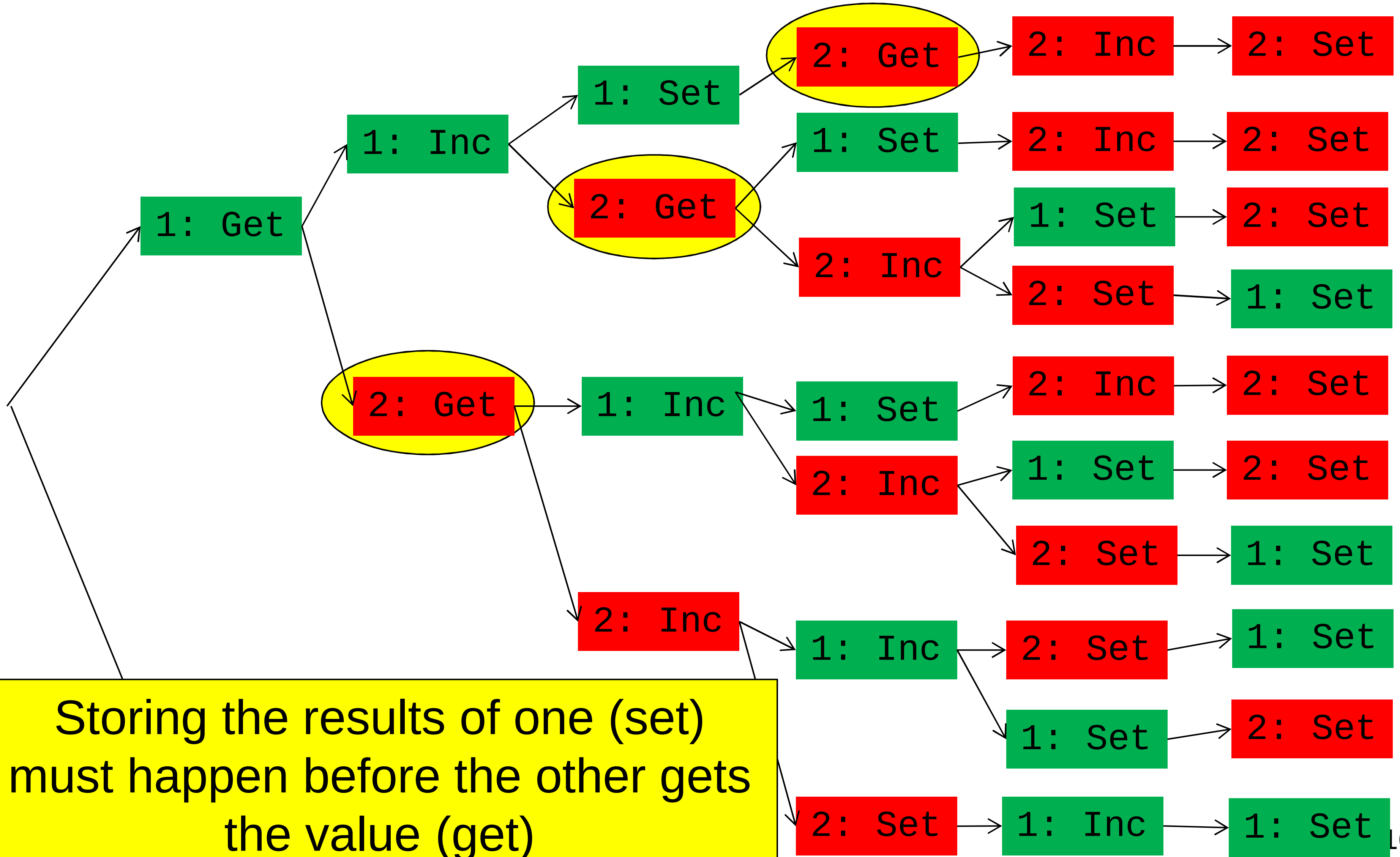
We will trace the potential execution orders for these operations to see whether the system will perform as we may expect

We will assume just one Get-Increment-Write for each, to simplify matters – the real situation is even worse

Potential traces



Which are good traces?

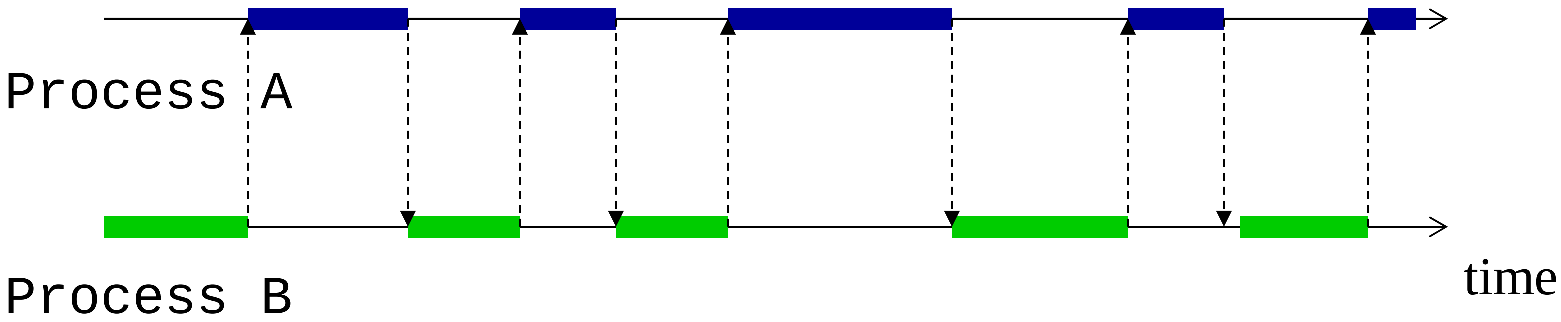


Storing the results of one (set) must happen before the other gets the value (get)

Methods to fix the problem

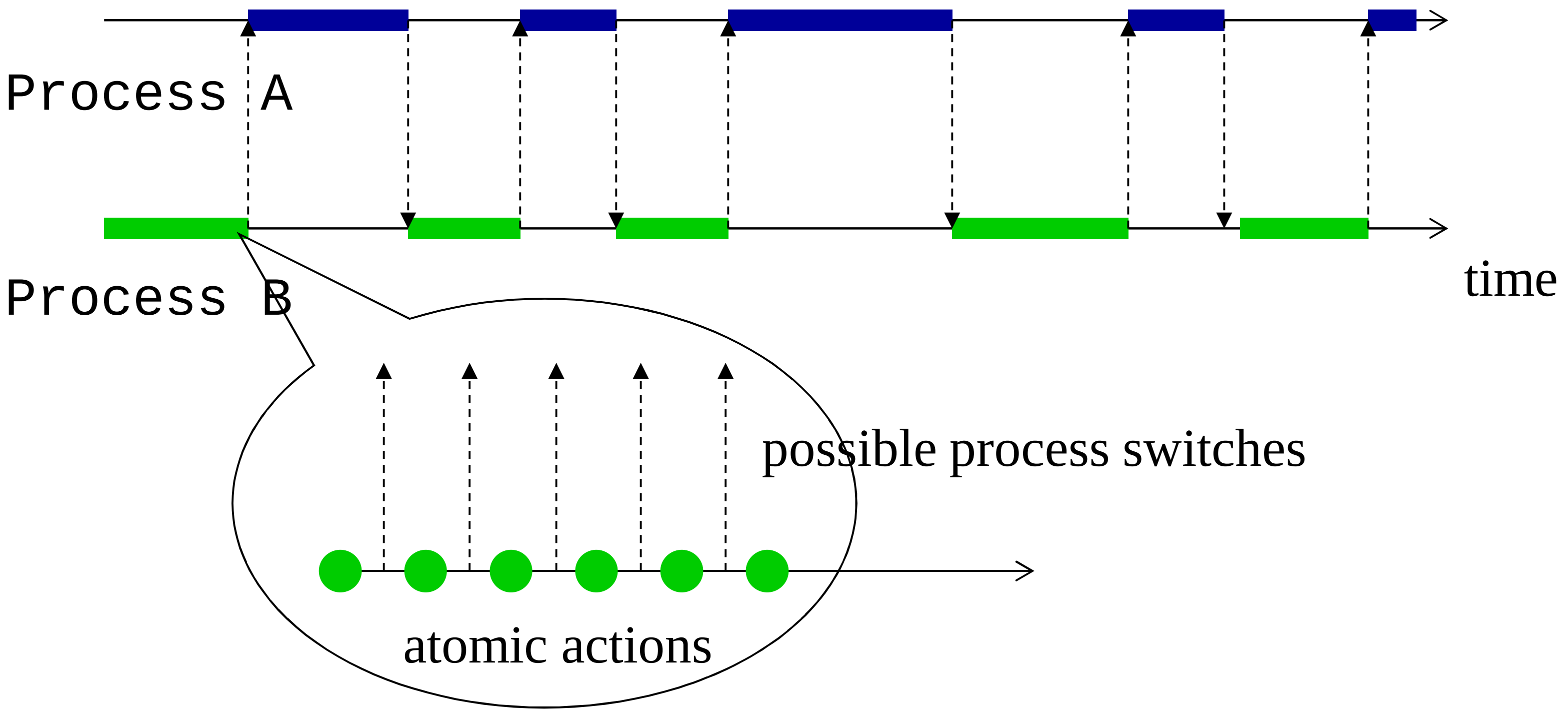
One processor (core)

- Consider a single processor situation
- Time is split between the processes



- At some point our process is interrupted

Process switches



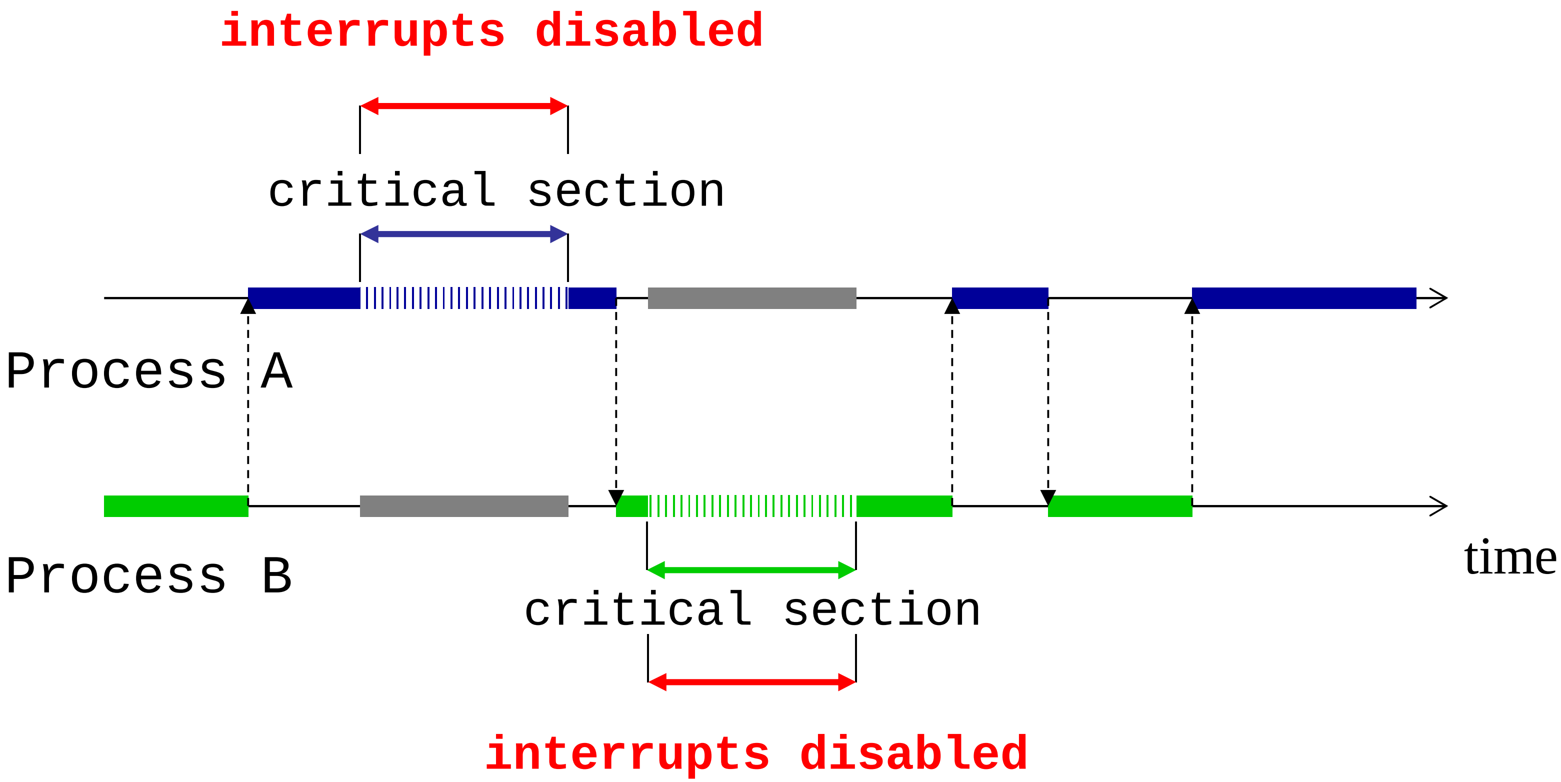
- Operations of a process consist of multiple atomic operations

Potential solution (single processor)

- The problem is that the get-increment-set is not an atomic operation

- Three obvious solutions on a single-processor system:
  1. Stop process switches happening between the operations
     - i.e. Turn off process switches/interrupts
  2. Make the get-increment-set operation atomic
     - Then it is just one operation
  3. Use hardware features (e.g. memory locks, transactional memory) – ignore for now

Turn off interrupts/switches



Problems with turning off interrupts

- Operating system must allow it
  – Makes the task scheduling harder
  – Usually very unwise
    - One process could take all of the CPU time if it doesn’t turn them on again
  – Some interrupts (e.g. from devices) may really need to be handled as soon as possible
  – System-level programming only usually
- In a multi-processor system it may not work
  – Interrupt disabling usually applies to same processor
  – Other threads can change things even when you prevent interrupts on your processor

Second possibility

- Make your operation atomic
  - i.e. the whole thing is then one operation and cannot be interrupted
- Most CPUs provide atomic operations which can avoid some of these problems
  - E.g. ‘test and set’, ‘increment’, ‘swap’
- Most C compilers allow you a facility to get at these atomic operations, e.g:
  - Interlocked API (windows, e.g. InterlockedIncrement)
  - __sync_... (gcc, e.g. __sync_fetch_and_add())
- Usually platform or compiler specific options!

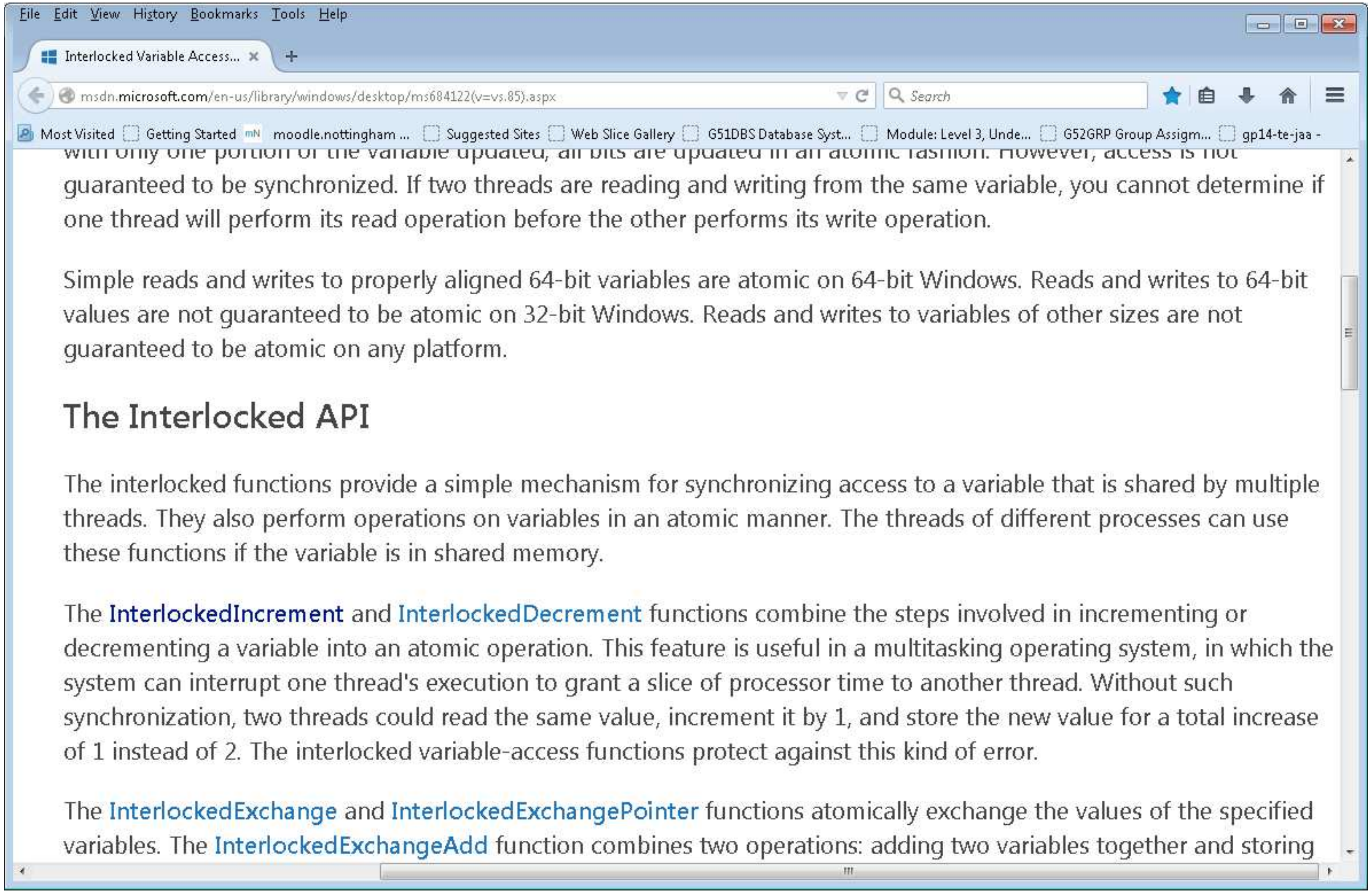
Atomic InterlockedIncrement

```
// Windows name for an unsigned long - 4 bytes
volatile DWORD dwTotal = 0;

// Function called by every thread
DWORD WINAPI thread_function( LPVOID lParam )
{
      for ( int i = 0; i < 1000000; i++ )
      {
            InterlockedIncrement(&dwTotal);
      }
      return 0;
}
```

Compiler-specific ‘atomic’ increment operation

Aside: Interlocked API

- http://msdn.microsoft.com/en-us/library/windows/desktop/ms684122%28v=vs.85%29.aspx



guaranteed to be synchronized. If two threads are reading and writing from the same variable, you cannot determine if one thread will perform its read operation before the other performs its write operation.

Simple reads and writes to properly aligned 64-bit variables are atomic on 64-bit Windows. Reads and writes to 64-bit values are not guaranteed to be atomic on 32-bit Windows. Reads and writes to variables of other sizes are not guaranteed to be atomic on any platform.

The Interlocked API

The interlocked functions provide a simple mechanism for synchronizing access to a variable that is shared by multiple threads. They also perform operations on variables in an atomic manner. The threads of different processes can use these functions if the variable is in shared memory.

The InterlockedIncrement and InterlockedDecrement functions combine the steps involved in incrementing or decrementing a variable into an atomic operation. This feature is useful in a multitasking operating system, in which the system can interrupt one thread's execution to grant a slice of processor time to another thread. Without such synchronization, two threads could read the same value, increment it by 1, and store the new value for a total increase of 1 instead of 2. The interlocked variable-access functions protect against this kind of error.

The InterlockedExchange and InterlockedExchangePointer functions atomically exchange the values of the specified variables. The InterlockedExchangeAdd function combines two operations: adding two variables together and storing

Problems with atomic operations

- Limited set of operations available
  - E.g. simple increment/decrement, test & set, compare and swap, etc
- What if you want to do something more complex?

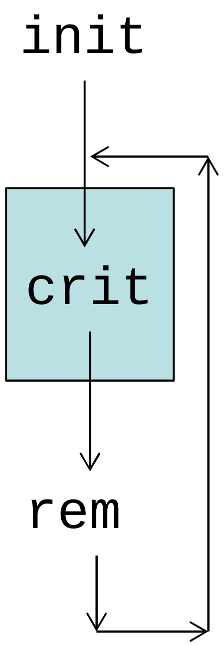
- Need a way to prevent two things doing something at the same time, regardless of what that thing is or how long it will take…

Critical sections

- Key element: no other thread must do something which affects what we do in the critical section
  - e.g. we must not let anything else get at the data while we are half-way through a change
- Obvious solutions:
  - Only do one operation – then either done or not, cannot be interrupted (atomic operations)
  - Take it in turns – check that no other thread will access the data while you want it
    - May be fair or not

Mutex / Critical Sections

- Simple mutual exclusion:
  - Mutually exclude each other – only one at a time
- We can think of the code having a ‘critical section’ that only one thread can run at once
- Consider a generic structure like this:
  1. Initial code (init) – before critical section
  2. Enter critical section – **apply protocol here**
  3. Critical section code (crit)
  4. Exit critical section – **apply protocol here**
  5. Remainder (rem) – after critical section



Simple round-robin

Variable: *turn*: integer variable, initialised to 1, **volatile**

- Thread 1:

init

Entry protocol:

while (*turn* != 1) ;

crit

Exit protocol:

turn = 2;

rem

- Thread 2:

init

Entry protocol:

while (*turn* != 2) ;

crit

Exit protocol:

turn = 1;

rem

Simple round-robin : trace (1)

Variable: *turn*: integer variable, initialised to 1, **volatile**

- Thread 1:

init

- Thread 2:

init

Simple round-robin : trace

Variable: *turn*: integer variable, initialised to 1, **volatile**

- Thread 1:

init

Entry protocol:

while (*turn* != 1) ;

- Thread 2:

init

Entry protocol:

while (*turn* != 2) ;

BLOCKED – effectively stops
Will not move until variable
***turn* changes value**

Simple round-robin : trace

Variable: *turn*: integer variable, initialised to 1, **volatile**

- Thread 1:

init

Entry protocol:

while (*turn* != 1) ;

crit

- Thread 2:

init

Entry protocol:

while (*turn* != 2) ;

Still blocked

Simple round-robin : trace

Variable: *turn*: **just got set to 2**

- Thread 1:

init

Entry protocol:

while (*turn* != 1) ;

crit

Exit protocol:

turn = 2;

- Thread 2:

init

Entry protocol:

while (*turn* != 2) ;

Just got unblocked

Simple round-robin : trace

Variable: *turn*: has value 2

- Thread 1:

init

Entry protocol:

while (*turn* != 1) ;

crit

Exit protocol:

turn = 2;

rem

- Thread 2:

init

Entry protocol:

while (*turn* != 2) ;

crit

Simple round-robin : trace

Variable: *turn*: has value 2

- Thread 1:

init

Entry protocol:

while (*turn* != 1) ;

crit

Exit protocol:

turn = 2;

rem

- Thread 2:

init

Entry protocol:

while (*turn* != 2) ;

crit

Assume that the crit takes a while

Simple round-robin : trace

Variable: *turn*: has value 2

- Thread 1:

init

Entry protocol:

while (*turn* != 1) ;

Now this one is blocked Cannot get any further yet

- Thread 2:

init

Entry protocol:

while (*turn* != 2) ;

crit

Assume that the crit is still taking a while

Simple round-robin : trace

Variable: *turn*: **just got set to 1 again**

- Thread 1:

init

Entry protocol:

while (*turn* != 1) ;

Eventually crit finishes in thread 2 and thread 1 can continue

- Thread 2:

init

Entry protocol:

while (*turn* != 2) ;

crit

Exit protocol:

turn = 1;

Simple round-robin : trace

Variable: *turn*: has value 1

- Thread 1:

init

Entry protocol:

while (*turn* != 1) ;

crit

- Thread 2:

init

Entry protocol:

while (*turn* != 2) ;

crit

Exit protocol:

turn = 1;

rem

and so on…

Simple round-robin

Variable: *turn*: integer variable, initialised to 1, **volatile**

- Thread 1:

init

Entry protocol:

while (*turn* != 1) ;

crit

Exit protocol:

turn = 2;

rem

Questions to consider:

- **When is each thread keeping the CPU busy?**
- Is this process fair across the threads?
  - Does each get a chance to act?
  - Does one get more actions than another?
- What happens if one thread stops?
- Can both be blocked at once?
- Can this be extended to more than 2?

Full round-robin algorithm

Variable: *turn*: integer variable, initialised to 1, **volatile**

- Thread 1:

init

Entry protocol:

while (*turn* != 1) ;

crit

Exit protocol:

turn = 2;

rem

- Thread 2:

init

Entry protocol:

while (*turn* != 2) ;

crit

Exit protocol:

turn = 1;

rem

Spin locks

- Sit in a tight loop waiting for a variable to take specific values
- Variable usually needs to be volatile
  - No point doing this if another thread is not going to alter it
- Thread will always be busy
  - Constantly checking the value
  - Wastes a lot of CPU time
- In the previous example they both had to take it in turns – we could do better…

Next lecture

- Mutual Exclusion
- Better entry/exit protocols for ensuring mutual exclusion and protecting critical sections