Chapter 26 Concurrency: An Introduction

So far we have considered

- A single physical CPU
- How to turn single CPU into *multiple virtual CPUs* (processes)
- Giving the illusion of multiple programs running at the same time

Thread: *new abstraction for a single process.*

Instead of our view of a single point of execution within a process;

That is, single Program Counter (PC) where instructions are fetched from.

Multi-threaded program: *more than one point of execution in a single process.*

Multiple PCs, each of which is being fetched from and executed.

Thread is like a process, except they share the same AS and can access the same data.

State of thread (similar to that of process)

- PC, SP, etc
- Private registers used for computation

Switching between threads: **context switch**

(Similar to switching between processes)

Process has: one or more **thread control blocks (TCBs)**

Instead of process control block (PCB)

Major difference:

- Switching between threads vs processes:
  - AS remains the same, so no need to switch PT.

Another difference:

- Multi-threaded process has multiple stacks (one per thread)

Example Fig 26.1: Now two stacks

- Any stack-allocated variables
- Parameters to functions
- Return values

So the AS is no longer “nice”…
But usually ok, since stacks generally don’t have to be so large.

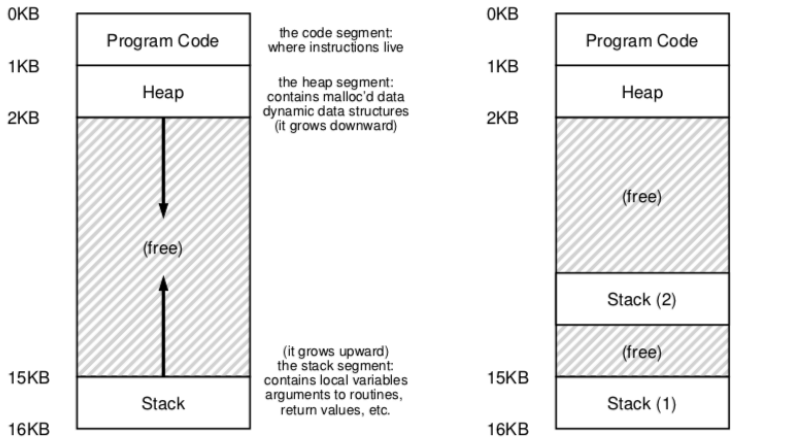


Figure 26.1: **Single-Threaded And Multi-Threaded Address Spaces**

Q: When will small stacks be a problem?

A: For programs that make heavy use of recursion!!

- Java used to have large fixed-sized per-thread stacks
- Go has small initial goroutine (~= thread) stacks that can grow dynamically as needed.

26.1 Why Use Threads?

First reason: **Parallelism**

Perform operations on very large arrays, e.g.,

- Increment the value of each element by some amount.
- Find word count of array of text

Can speed up this process by letting multiple CPUs do the work

Transforming standard single-threaded program to take advantage of multiple CPUs is called **parallelization**.

Second reason: **Avoid blocking progress due to slow I/O**

Allow other thread to continue when some thread is blocked on I/O.

- Waiting to send/receive a message
- Waiting for disk I/O to complete

Instead of waiting:

- Another thread can utilize CPU or
- Issue other I/O requests

Threading enables overlapping I/O with other activities within a single program.

Many server-based applications make use of threads in their implementation

- Web server
- Database management systems

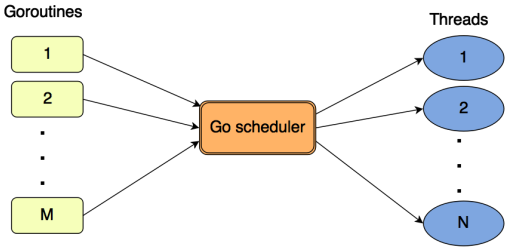
Could use multiple processes instead of threads for these cases

- But threads share AS and makes it easy to share data between threads
- Processes are more suitable for
  - Logically separate tasks
  - With little sharing of in-memory data structures

26.2 An Example: Thread Creation

Live coding: Fig 26.2 converted to Go;

We map: threads => goroutines



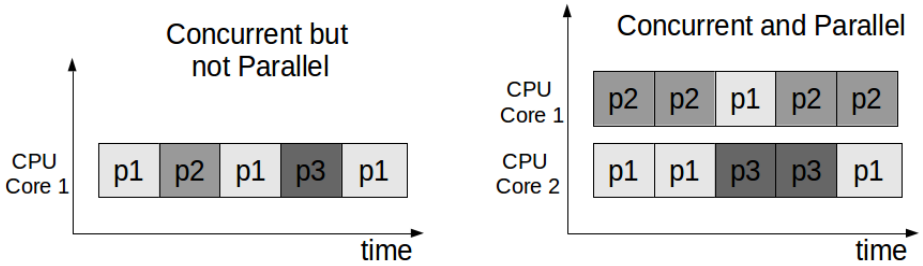
Now:

- Instead of executing program sequentially
- Newly created threads run independently of the main thread

The OS scheduler decides which thread gets to run.

We say that a thread may run concurrently with another thread.

Note that **concurrency is not parallelism.**



main	Thread 1	Thread2
starts running		
prints “main: begin”		
creates Thread 1		
creates Thread 2		
waits for T1		
	runs	
	prints “A”	
	returns	
waits for T2		
		runs
		prints “B
		returns
prints “main: end”		

Figure 26.3: **Thread Trace (1)**

main	Thread 1	Thread2
starts running		
prints “main: begin”		
creates Thread 1		
	runs	
	prints “A”	
	returns	
creates Thread 2		
		runs
		prints “B”
		returns
waits for T1		
returns immediately; T1 is done		
waits for T2		
returns immediately; T2 is done		
prints “main: end”		

Figure 26.4: **Thread Trace (2)**

main	Thread 1	Thread2
starts running		
prints “main: begin”		
creates Thread 1		
creates Thread 2		
		runs
		prints “B”
		returns
waits for T1		
	runs	
	prints “A”	
	returns	
waits for T2		
returns immediately; T2 is done		
prints “main: end”		

Figure 26.5: **Thread Trace (3)**

26.3 Why it Gets Worse: Shared Data

Example thread execution traces are not sharing any data

Not a problem if each thread is completely independent — don't share any data.

Live coding Fig 26.6 converted to Go.

26.4 The Heart of the Problem: Uncontrolled Scheduling

The instructions to increment the counter variable looks like this:

```
mov   0x0849a1c, %eax
add   $0x01, %eax
mov   %eax, 0x0849a1c
```

- Load variable counter from memory address 0x0849a1c into register %eax
- Add 1 to the register %eax
- Save register value to counter's memory location 0x0849a1c

Example Fig 26.7.

- T1 runs first two instructions; timer interrupt goes off
- T2 runs all three instructions

Since T1 didn't save the update 51 to memory
T2 will also load 50 and update to 51.
And when T1 saves its register to memory it too will update to 51.

Correct version of the program should give the result 52.

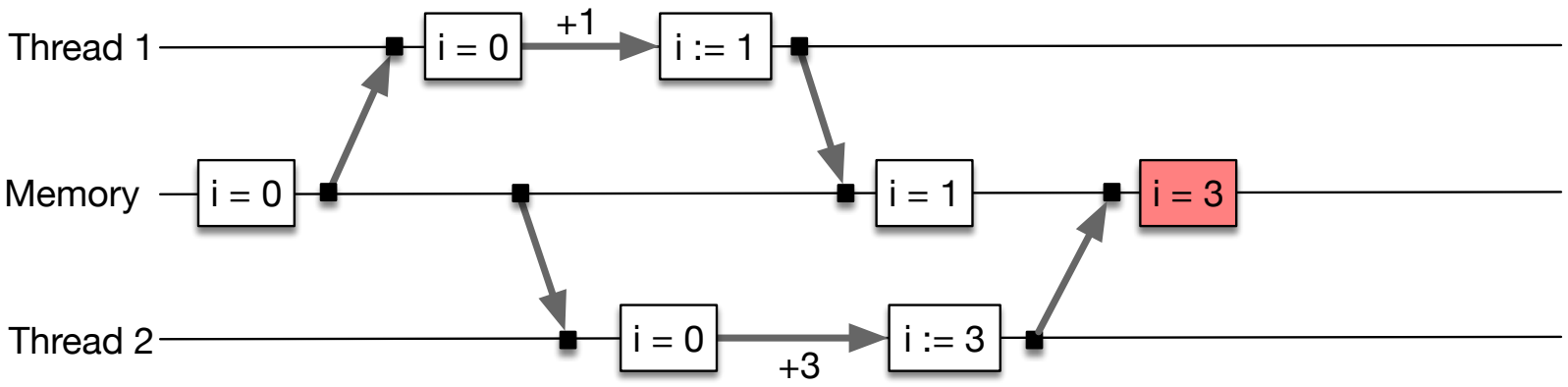
			(after instruction)		
OS	**Thread 1**	**Thread 2**	**PC**	**%eax**	**counter**
	before critical section		100	0	50
	mov 0x8049a1c, %eax		105	**50**	50
	add $0x1, %eax		108	**51**	50
interrupt					
save T1's state					
restore T2's state			100	0	50
		mov 0x8049a1c, %eax	105	**50**	50
		add $0x1, %eax	108	**51**	50
		mov %eax, 0x8049a1c	113	51	**51**
interrupt					
save T2's state					
restore T1's state			108	51	51
	mov %eax, 0x8049a1c		113	51	**51**

Figure 26.7: **The Problem: Up Close and Personal**

This is called a **race condition** or a **data race**.

- Results depend on the timing execution of the code
- Can get different results each time: **indeterminate**
- We want **deterministic** computation

Example of a data race.



Def. Critical Section: a piece of code that *accesses* (read or write) a shared variable or resource and must not be concurrently executed by more than one thread.

A critical section is code that when executed by multiple threads may result in a ***race condition****.*

What we want:

Def. Mutual Exclusion: guarantee that if one thread is executing within the critical section, other threads are prevented from doing so!

Edsger Dijkstra

- coined these terms
- Pioneer in the field

26.5 The Wish for Atomicity

HW: more powerful instructions to do this as a single step without untimely interrupt

```
memory-add          0x8049a1c, $0x01
```

HW should guarantee that it executes **atomically.**

Def. Atomically: As as unit (all or none)

What we'd like is to execute the three instructions atomically.

```
mov    0x0849a1c, %eax
add    $0x01, %eax
mov    %eax, 0x0849a1c
```

Not practical to add atomic update instructions for all instruction types.

Instead: HW should provide a few useful/general **synchronization primitives**.

We use these primitives to ensure multi-threaded code

- Accesses critical sections in a synchronized and controlled manner
- Reliably produce correct results
- Despite challenging nature of concurrent execution

These are hard problems!

Summary

We can use such primitives to ensure multi-threaded code

- Accesses critical sections in a synchronized and controlled manner
- Reliably produce correct results
- Despite challenging nature of concurrent execution

- These are hard problems!
- Difficult to use synchronization primitives correctly!

Series of actions that are atomic, means all or nothing.

Sometimes called transactions.

Def. Transaction: *grouping of many actions into a single atomic action.*