Go Preemptive Scheduler Design Doc

Dmitry Vyukov
dvyukov@
May 15, 2013

Problem

1. Some user programs visibly "misbehave" due to absence of preemption, a single goroutine can run all the time and do not switch to other goroutines, timers do not fire.
2. Currently GC needs to "stop the world", and w/o means for preemption this can take arbitrary amount of time (users reported up to several minutes).
Examples of the issues:
https://code.google.com/p/go/issues/detail?id=5324
https://code.google.com/p/go/issues/detail?id=4711
https://code.google.com/p/go/issues/detail?id=5163
https://code.google.com/p/go/issues/detail?id=543

Proposed Solution

The idea is to use existing split stack mechanism for preemption. Each goroutine checks for split stack on every function call, if it's out of stack, it calls into runtime. If we can make this check fail by an external request, we get the preemption.
Sysmon thread watches for P's executing the same goroutine for more than X ms. If finds a one it sets g->stackguard to a very large value, so that on next split stack check the goroutine calls morestack(). Morestack() is modified to check for preemption and call into scheduler if needed.
GC issues preemption request for all currently running goroutines during stop the world.

It should plays well with precise GC because goroutine are preempted only at controllable points. So we can emit all the necessary information required for GC.

Pros:
- No additional overheads
- Relatively simple implementation, no signals, minimal synchronization

Cons:
- Increases runtime complexity
- ?

Evaluation

For evaluation I've used the prototype of the design:
https://codereview.appspot.com/9136045

Go1 benchmark suite shows no slowdown:

benchmark	old ns/op	new ns/op	delta
BenchmarkBinaryTree17	4604208550	4607734202	+0.08%
BenchmarkFannkuch11	2903981794	2925657766	+0.75%
BenchmarkFmtFprintfEmpty	83	82	-1.44%
BenchmarkFmtFprintfString	210	210	+0.00%
BenchmarkFmtFprintfInt	171	168	-1.75%
BenchmarkFmtFprintfIntInt	284	271	-4.58%
BenchmarkFmtFprintfPrefixedInt	259	260	+0.39%
BenchmarkFmtFprintfFloat	377	377	+0.00%
BenchmarkFmtManyArgs	1067	1053	-1.31%
BenchmarkGobDecode	8793949	8781962	-0.14%
BenchmarkGobEncode	10055812	10144175	+0.88%
BenchmarkGzip	422396880	413429581	-2.12%
BenchmarkGunzip	100201142	99900778	-0.30%
BenchmarkHTTPClientServer	43658	43523	-0.31%
BenchmarkJSONEncode	34239405	33973689	-0.78%
BenchmarkJSONDecode	79118008	77890662	-1.55%
BenchmarkMandelbrot200	4033471	4034173	+0.02%
BenchmarkGoParse	5209448	5256012	+0.89%
BenchmarkRegexpMatchEasy0_32	106	107	+0.94%
BenchmarkRegexpMatchEasy0_1K	301	300	-0.33%
BenchmarkRegexpMatchEasy1_32	89	90	+0.67%
BenchmarkRegexpMatchEasy1_1K	755	749	-0.79%
BenchmarkRegexpMatchMedium_32	163	163	+0.00%
BenchmarkRegexpMatchMedium_1K	59182	58977	-0.35%
BenchmarkRegexpMatchHard_32	2796	2810	+0.50%
BenchmarkRegexpMatchHard_1K	91888	92296	+0.44%
BenchmarkRevcomp	685704524	687030150	+0.19%
BenchmarkTemplate	111448907	111908050	+0.41%
BenchmarkTimeParse	408	405	-0.74%
BenchmarkTimeFormat	437	438	+0.23%

The following program demonstrates the GC "stop the world" issue:
http://play.golang.org/p/Yzc4Vx-KaF

Current GC trace:

```
gc7(8): 0+0+429 ms, 3462 -> 2908 MB
gc8(8): 0+0+296 ms, 5830 -> 3861 MB
gc9(8): 0+0+661 ms, 7758 -> 3825 MB
gc10(8): 0+0+939 ms, 7664 -> 4014 MB
gc11(8): 0+0+907 ms, 8063 -> 4016 MB
```

GC trace with the preemptive scheduler:

```
gc8(8): 0+0+126 ms, 4989 -> 3020 MB
gc9(8): 0+0+124 ms, 6057 -> 3249 MB
gc10(8): 0+0+72 ms, 6499 -> 3711 MB
gc11(8): 0+0+121 ms, 7434 -> 3250 MB
```

Note significant decrease in the stoptheworld pause.

The following test does not work now, but works with the preemptive scheduler:
http://play.golang.org/p/86i_dRxWBm

Implementation Plan

1. Introduce a copy of g->stackguard variable, because it can be overwritten during preemption.
2. Sysmon background thread watches for goroutines that execute for more than X ms (the initial proposed value is 10 ms) stores preemption mark into g->stackguard.
3. GC issues preemption request for all running g's.
4. Morestack() checks for preemption mark and switches the goroutine if possible.
5. Protect critical sections in runtime with m->lock++/-- (e.g. runtime.newproc, runtime.ready). See Non-preemptible Regions section below.
6. Properly synchronize finalizer goroutine with GC, because it can not longer rely on preemption at known points.
7. Remove existing runtime.gcwaiting checks in chan/hashmap/malloc (poor man's preemption).

At this point we get working preemptive scheduler.

8. Currently framesize is not always passed to morestack() to save code size. It makes it impossible to reuse the current stack frame after preemption (even unsuccessful), and forces to allocate a new frame each time.
The proposed solution is to introduce morestackNxM() functions, where N is argsize (8,16..64) and M - framesize (8,16..64), i.e. 64 functions; and a general morestack() function that accepts argsize and framesize explicitly. This way we will always know argsize/framesize in morestack(), and so will be able to reuse frames. The exact ranges for N and M require additional investigation, potentially less than 64 functions is sufficient.

9. Refactor gogo/gogocall/gogocallfn. We have 3 of them, and context restoration after preemption requires one more (gogo that restores DX -- closure context). We can have 1 context switching function that accepts and restores both AX and DX. The proposed interface is:

```
// "Executes" PUSH PC in the BUF context.
void runtime·returnto(Gobuf *BUF, uintptr PC);
```

```
// Moves CRET into AX, CTX into DX and switches to BUF.
void runtime·gogogo(Gobuf *BUF, uintptr CRET, uintptr CTX);
```

The existing functions and the new function gogo2 can be implemented in terms of the interface as follows:

```
void runtime·gogo(Gobuf *buf, uintptr cret)
{
	runtime·gogogo(buf, cret, 0);
}
void runtime·gogocall(Gobuf *buf, void(*f)(void), uintptr ctx)
{
	runtime·returnto(buf, buf.pc);
	buf.pc = f;
	runtime·gogogo(buf, 0, ctx);
}
void runtime·gogocallfn(Gobuf *buf, FuncVal *fn)
{
	runtime·returnto(buf, buf.pc);
	buf.pc = *(uintptr*)fn;
	runtime·gogogo(buf, 0, fn);
}
void runtime·gogo2(Gobuf *buf, uintptr ctx)
{
	runtime·gogogo(buf, 0, ctx);
}
```

Points 1-9 are implemented before Go1.2.

10. Collect experience with the scheduler and decide on necessity of compiling additional preemption checks on back edges. Checks on function entry should be sufficient for most practical cases, so it's unclear whether checks on back edges are required. The check may look as:

```
MOV	[g], CX
CMP	$-1, g_stackguard(CX)
JNZ	nopreempt
CALL	$runtime.preempt(SB)
nopreempt:
…
```

11. An optimization proposed by iant@ is to allocate an additional TLS slot for stackguard. This

will allow to optimize split stack checks and the additional preemption checks:

```
CMP    $-1, [stackguard]
JNZ    nopreempt
CALL   $runtime.preempt(SB)  // can be further moved onto cold path
nopreempt:
```

Points 10, 11 are not implemented for Go1.2.

Non-preemptible Regions

The preemptive scheduler adds a new complication to the runtime library -- a goroutine can be preempted and descheduled at a lot more points. Current code is not ready for this.
One of the measures to mitigate this is to do very conservative preemption. Namely a goroutine is not preempted if one of the following is true:
- it holds runtime locks
- it executes on g0
- it's mallocing or gcing
- it's not in Grunning state (e.g. Gsyscall)
- it does not have a P or the P is not in Prunning state

This covers most of the cases where preemption is unwanted. However, there are some remaining cases. I've identified 2 places in the scheduler: runtime.newproc() and runtime.ready(), in both cases a goroutine can hold a P in a local variable; it's just bad, and leads to deadlock if stoptheworld is requested (the P will never be stopped).
Chans are protected by locks, and hashmaps seems to be safe.
The general recipe is: the preemption must be disabled when shared data structures (e.g. chans, hashmaps, scheduler, memory allocator, etc) are in inconsistent state, and that inconsistency can break either scheduler or GC.
The proposed mechanism to manually disable preemption is to use m->lock++/--. It's already used to disable GC and preemption in runtime.