Concurrency and Lightweight Threads

Mainack Mondal
Sandip Chakraborty

CS60203
Autumn 2024

Today's class

- **Parallelism and concurrency**
- What's the issue with OS threads?
    - Overhead, and latency
- How to implement concurrency? Event loops
    - Python asyncio event loop
- Extending to multiple cores
    - Motivating work stealing
    - Go runtime internals

Parallelism and concurrency

Parallelism

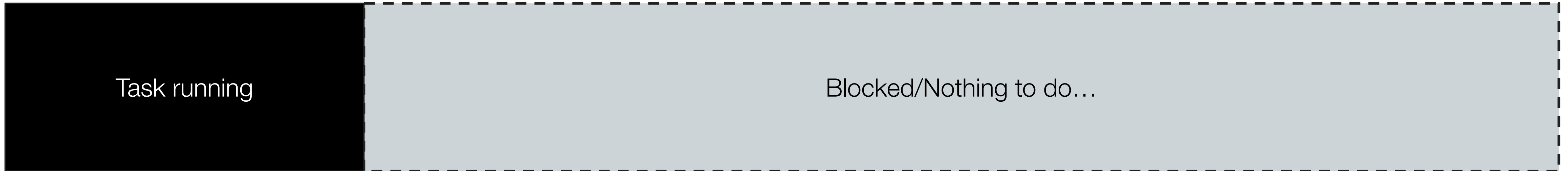
- Parallelism is when **tasks literally run at the same time**, e.g., on a multicore processor.
  - So far we have seen operating systems threads used for parallelising tasks
  - In terms of threads, Sun defines parallelism as- A condition that arises when at least two threads are executing simultaneously.
  - We have also seen ILP, and SIMD parallelism

Parallelising **CPU bound tasks** improves **throughput**

But what if, tasks are idle most of the time?

Does parallelising it help?

What if task is idle/blocked most of the time?



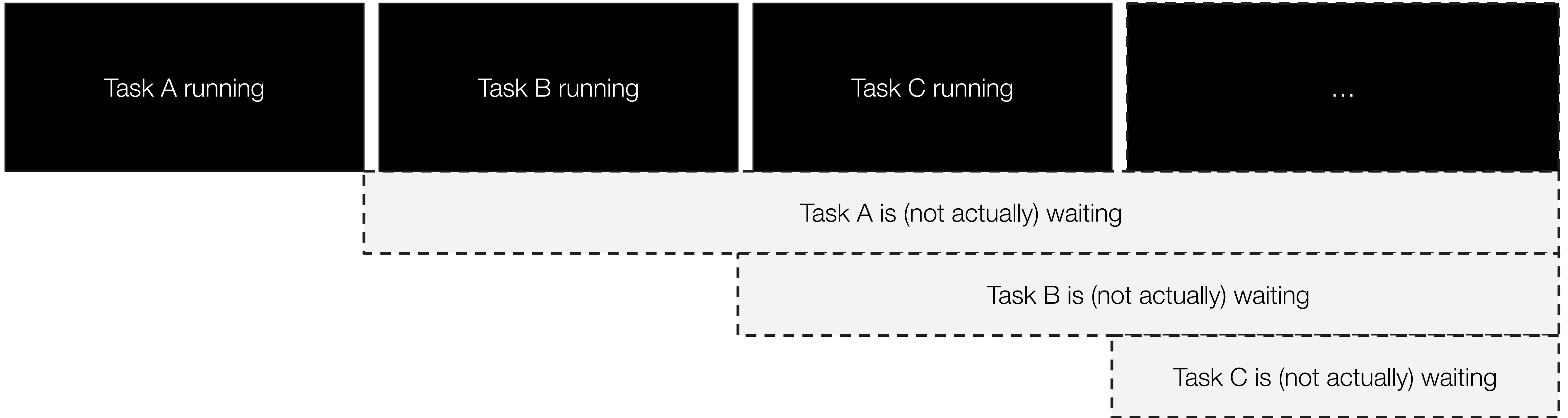
This is very common…

- Databases, and logging systems are often blocked on disk IO
- HTTP server handling a request is often blocked on network IO
- Data pipelines are blocked on previous stages
- ….

What if you parallelise it?

It gets a bit better… but clearly this can't be the best we can do

What if…you could do this



This is the **idea of concurrency**

Concurrency

- Concurrency is when two or more tasks can **start, run, and complete together**
  - It doesn't necessarily mean they'll ever both be running at the same instant.
  - For example, **multitasking on a single-core machine**
  - Sun defines concurrency as- A condition that exists when at least two threads are making progress. A more generalized form of parallelism that can include time-slicing as a form of virtual parallelism.

Concurrently performing **IO bound tasks** improves **throughput**

Isn’t this how the OS scheduling (and IO) works already?

Yes it is :)

But there is a catch

Today's class

- Parallelism and concurrency
- **What's the issue with OS threads?**
  - **Overhead, and latency**
- How to implement concurrency? Event loops
  - Python asyncio event loop
- Extending to multiple cores
  - Motivating work stealing
  - Go runtime internals

Overhead in OS threads

Overhead in OS threads

Starting up an OS thread, or even **waking it up** takes **many clock cycles**

```
auto mythread = std::thread([] { counter++; });
mythread.join();
```

How much time do you think it will take per thread?

Overhead in OS threads

Starting up an OS thread, or even waking it up takes many clock cycles

```
auto mythread = std::thread([] { counter++; });
mythread.join();
```

On my machine (Core i9)-

50202.5 ns +/- 23485.3 (average +/- std. deviation)

min: 35700

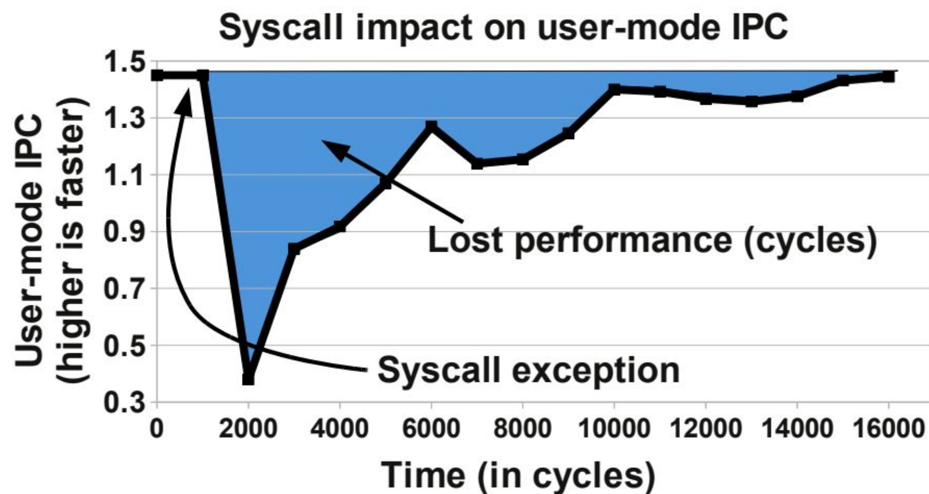
max: 248268

It's taking thousands of clock cycles!

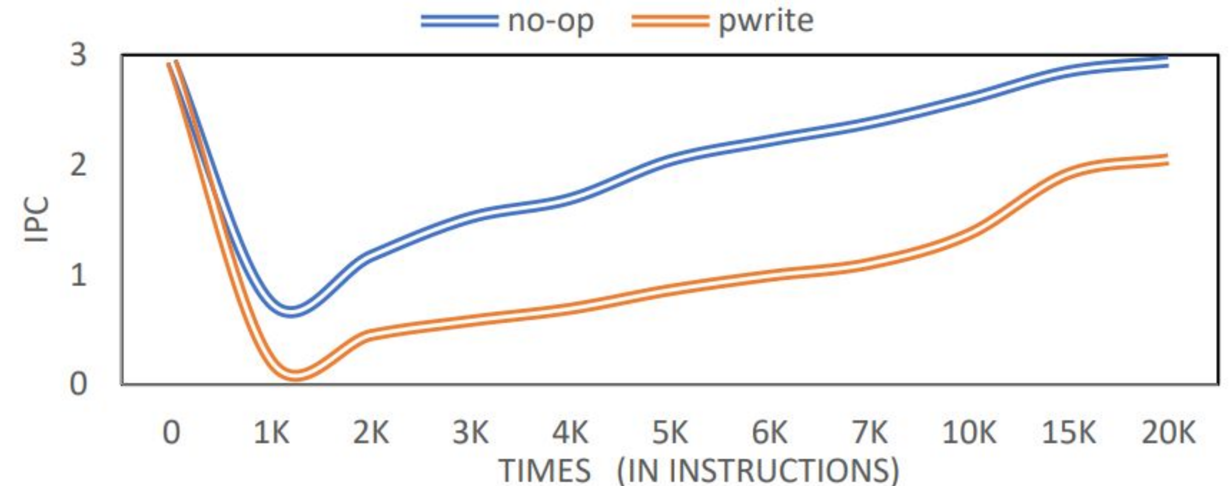
Recall: But…how bad is a context switch?

Userspace **IPC drops to around 0 right after syscall**, and takes **~20k instructions** to come back to normal

`Performance loss (Soares et al., 2010)`



`Performance loss (Zhou et al., 2023)`



Overhead in OS threads

Starting up an OS thread, or even waking it up takes many clock cycles

1. Hence OS threads **should not be used** when **unblocked compute time is small, and is overshadowed by creation/start-up time**

This is clearly the case with concurrent programs!

Overhead in OS threads

Starting up an OS thread, or even waking it up takes many clock cycles

1. Hence, OS threads should not be used when compute time is small, and is overshadowed by creation/start-up time
2. Hence, OS threads should also **not be used in places** where their **results are to be used in control flow**

This is because control flow of the thread in question will start to block, and **incur latency of thread switches!**

Overhead in OS threads

Starting up an OS thread, or even waking it up takes many clock cycles

1. Hence, OS threads should not be used when compute time is small, and is overshadowed by creation/start-up time
2. Hence, OS threads should also not be used in places where their results are to be used in control flow
3. OS threads are **allocated larger stack memory**, and **kernel does not have an option to decommit allocated memory**

But, given that paging is on-demand, do you think that is an issue?

Today's class

- Parallelism and concurrency
- What's the issue with OS threads?
    - Overhead, and latency
- **How to implement concurrency? Event loops**
    - **Python asyncio event loop**
- Extending to multiple cores
    - Motivating work stealing
    - Go runtime internals

How to implement concurrency? Event loops

How to do things concurrently?

- So far we have seen that **OS threads have overheads**, which become a bottleneck for highly concurrent systems

- Here we will try to implement **concurrency from first principles**, and notice a pattern

- This pattern is termed, an **“Event Loop”**, and is commonly used to develop concurrent systems

The event loop - Wikipedia

In computer science, the event loop (also known as message dispatcher, message loop, message pump, or run loop) is a programming construct or design pattern that **waits for and dispatches events or messages in a program**.

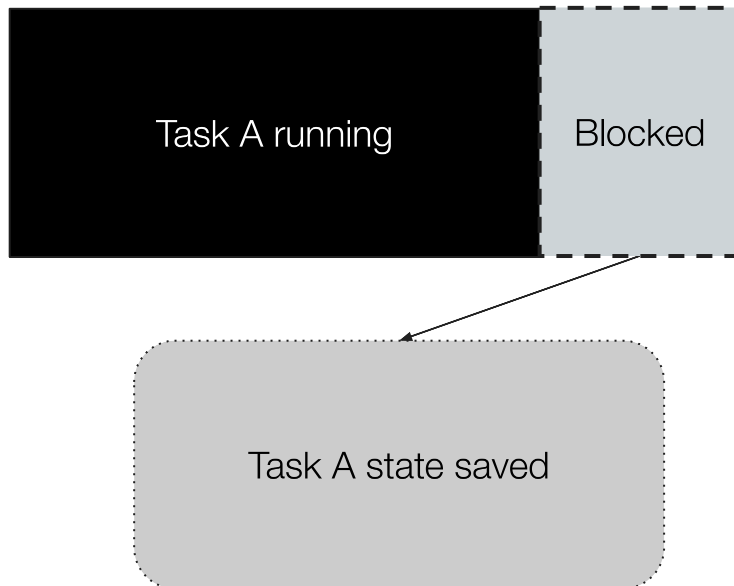
The event loop works by **making a request to some internal or external "event provider"** (that generally blocks the request until an event has arrived), then calls the relevant event handler ("dispatches the event").

How to implement concurrency? Step-1

1. **Start a task A**, and do required computation

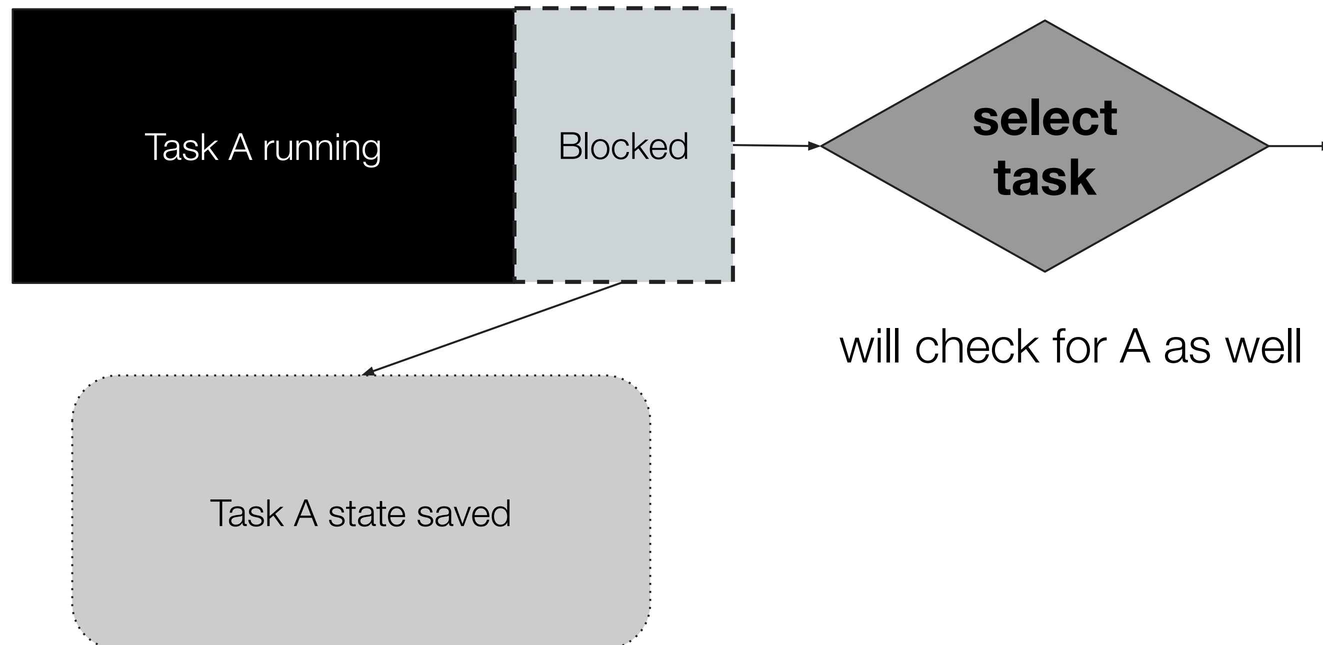
How to implement concurrency? Step-2

1. Start a task A, and do required computation
2. When task A blocks/waits, **save state of A**



How to implement concurrency? Step-3

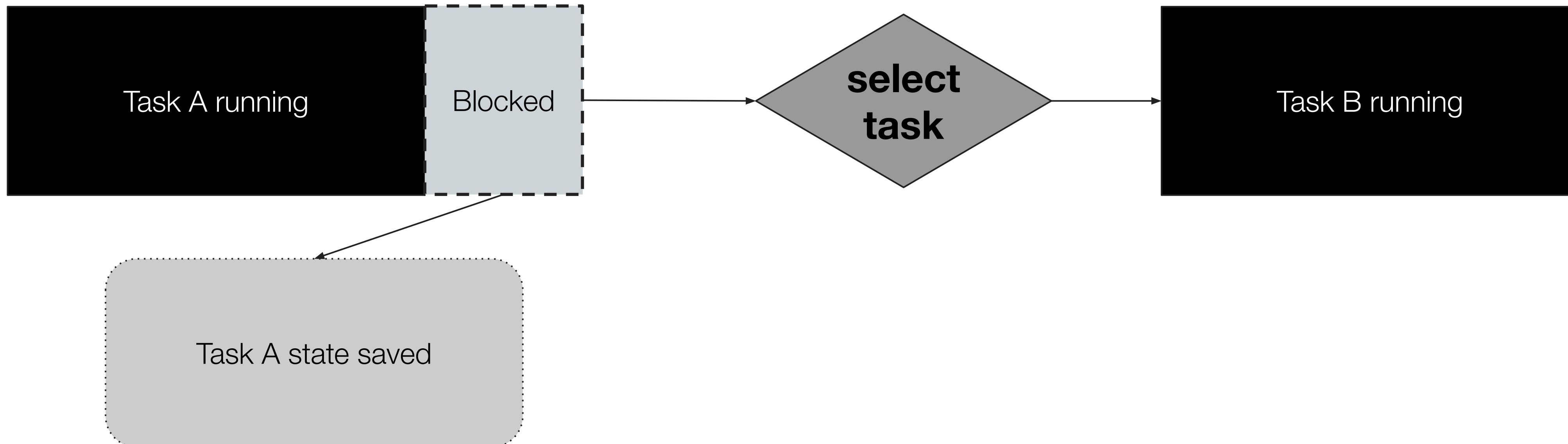
1. Start a task A, and do required computation
2. When task A blocks/waits, save state of A
3. **Add check** in selector to see if A is unblocked, **i.e. ready** to run again



will check for A as well

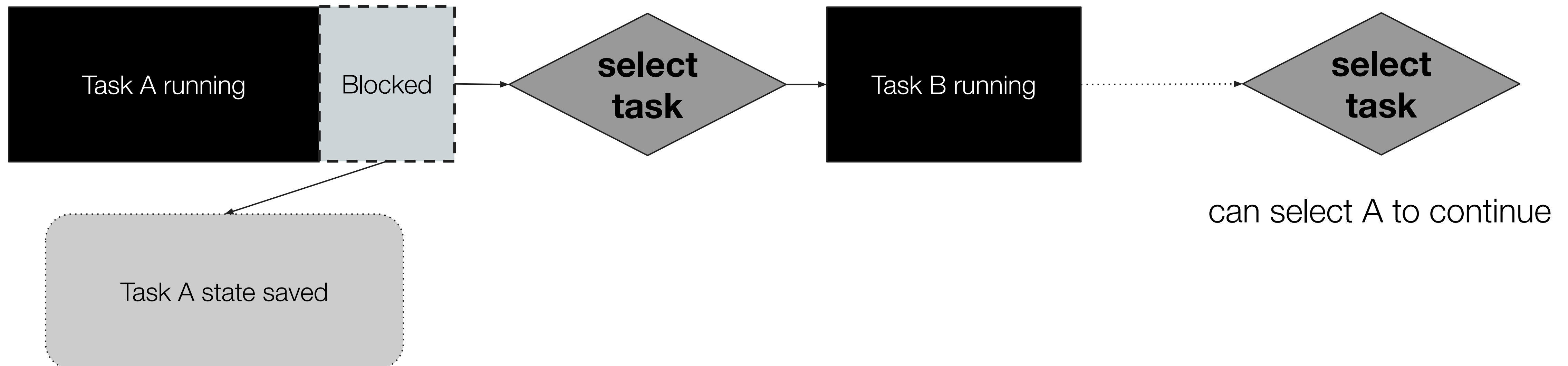
How to implement concurrency? Step-4

1. Start a task A, and do required computation
2. When task A blocks/waits, save state of A
3. Add check in selector to see if A is unblocked, i.e. ready to run again
4. **Check for "ready tasks", and select one** among them **to run**



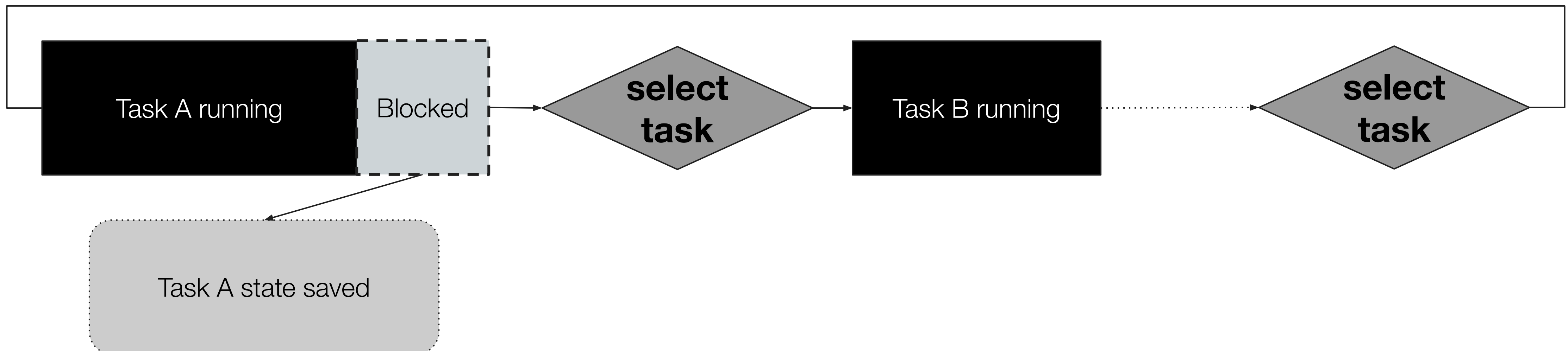
How to implement concurrency? Step-4

1. Start a task A, and do required computation
2. When task A blocks/waits, save state of A
3. Add check in selector to see if A is unblocked, i.e. ready to run again
4. **Check for "ready tasks", and select one** among them **to run**



How to implement concurrency? Step-5

1. Start a task A, and do required computation
2. When task A blocks/waits, save state of A
3. Add check in selector to see if A is unblocked, i.e. ready to run again
4. Check for “ready tasks”, and select one among them to run
5. **Repeat for each task!** (yep this is the event loop)

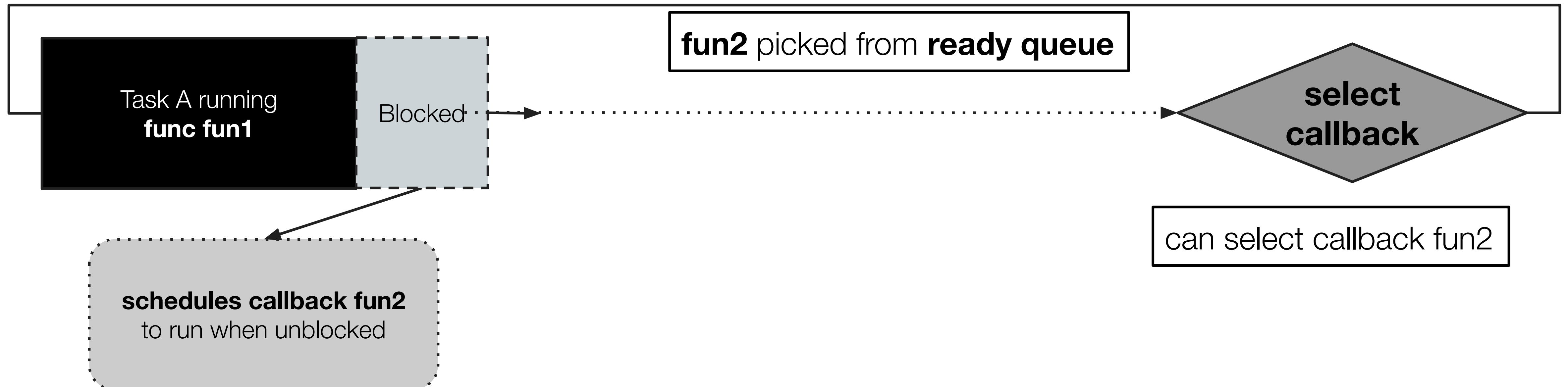


Callbacks

To implement such a design, we use the idea of callbacks. A callback is a piece of code (often a function), used to schedule as a task on the event-loop.

Figure sequence:

`func fun1` → `schedules callback fun2` → `loop calls callback fun2 when unblocked`



Event loops and callbacks

1. Start a task A, and do required computation
2. When task A blocks/waits, save state of A
3. Add check in selector to see if A is unblocked, if so it can be run again
4. Select another task to run and run it
5. Repeat for each task! (yep this is the loop)

At a high level, we need to **define an event loop** that does following-

```
while true:
   1. Select callback to run from ready queue
   2. Run the callback
   3. if needed: Schedule a new callback
```

Python `asyncio`

A real world event loop

asyncio — Asynchronous I/O

asyncio is a library to write **concurrent** code using the **async/await** syntax.

asyncio is used as a foundation for multiple Python asynchronous frameworks that provide high-performance network and web-servers, database connection libraries, distributed task queues, etc.

asyncio is often a perfect fit for IO-bound and high-level **structured** network code.

Hello World!

```
import asyncio

async def main():
    print('Hello ...')
    await asyncio.sleep(1)
    print('... World!')

asyncio.run(main())
```

asyncio provides a set of **high-level** APIs to:

- run Python coroutines concurrently and have full control over their execution;
- perform network IO and IPC;
- control subprocesses;
- distribute tasks via queues;
- synchronize concurrent code;

Additionally, there are **low-level** APIs for *library and framework developers* to:

- create and manage event loops, which provide asynchronous APIs for networking, running subprocesses, handling OS signals, etc;
- implement efficient protocols using transports;
- bridge callback-based libraries and code with async/await syntax.

asyncio — Asynchronous I/O — Python 3.12.5 documentation

The event loop definition

A reference implementation of the event loop can be taken from

`class BaseEventLoop`

From [cpython/Lib/asyncio/base_events.py at main]

```
def run_forever(self):
    """Run until stop() is called."""
    try:
        self._run_forever_setup()
        while True:
            self._run_once()
            if self._stopping:
                break
    finally:
        self._run_forever_cleanup()
```

The loop

one iteration

`_run_once`: One iteration of the loop

1. Performs bookkeeping on delayed/cancelled tasks
2. Maintains `self._ready` queue of tasks that can be run
   - tasks **scheduled to be called later**

```
# Handle 'later' callbacks that are ready.
end_time = self.time() + self._clock_resolution
while self._scheduled:
    handle = self._scheduled[0]
    if handle._when >= end_time:
        break
    handle = heapq.heappop(self._scheduled)
    handle._scheduled = False
    self._ready.append(handle)
```

`_run_once`: One iteration of the loop

1. Performs bookkeeping on delayed/cancelled tasks
2. Maintains `self._ready` queue of tasks that can be run
   - tasks scheduled to be called later
   - tasks **waiting on IO** (using a selector)

Note- selector is chosen on the basis of what is faster for OS, like `epoll` for Linux

```
event_list = self._selector.select(timeout)
self._process_events(event_list)
# Needed to break cycles when an exception occurs.
event_list = None
```

`_run_once`: One iteration of the loop

1. Performs bookkeeping on delayed/cancelled tasks
2. Maintains `self._ready` queue of tasks that can be run
   - tasks scheduled to be called later
   - tasks waiting on IO (using a selector)
3. Executes the tasks in `self._ready` using `handle._run()`

runs task

```
for i in range(ntodo):
    handle = self._ready.popleft()
    if handle._cancelled:
        continue
    if self._debug:
        try:
            self._current_handle = handle
            t0 = self.time()
            handle._run()
```

```
            dt = self.time() - t0
            if dt >= self.slow_callback_duration:
                logger.warning('Executing %s took %.3f seconds',
                               _format_handle(handle), dt)
        finally:
            self._current_handle = None
    else:
        handle._run()
```

Today's class

- Parallelism and concurrency
- What's the issue with OS threads?
  - Overhead, and latency
- How to implement concurrency? Event loops
  - Python asyncio's event loop
- **Extending to multiple cores**
  - **Motivating work stealing**
  - **Go runtime internals**

Extending to multiple cores

The m:n problem

So far we have been running instructions on one processor, or to be exact, one OS thread

The idea of user-level concurrency can also be applied to more than one OS threads

The general version is about running **m virtual threads, on n OS level threads**

Attempt 1

Event loop with a thread pool

Attempt 1: Event loop with a thread pool

1. The idea is to extend the **event loop for each thread** in a thread-pool.

2. We maintain a **central ready queue**, and each thread can pick from it

Recall: Event loops and callbacks

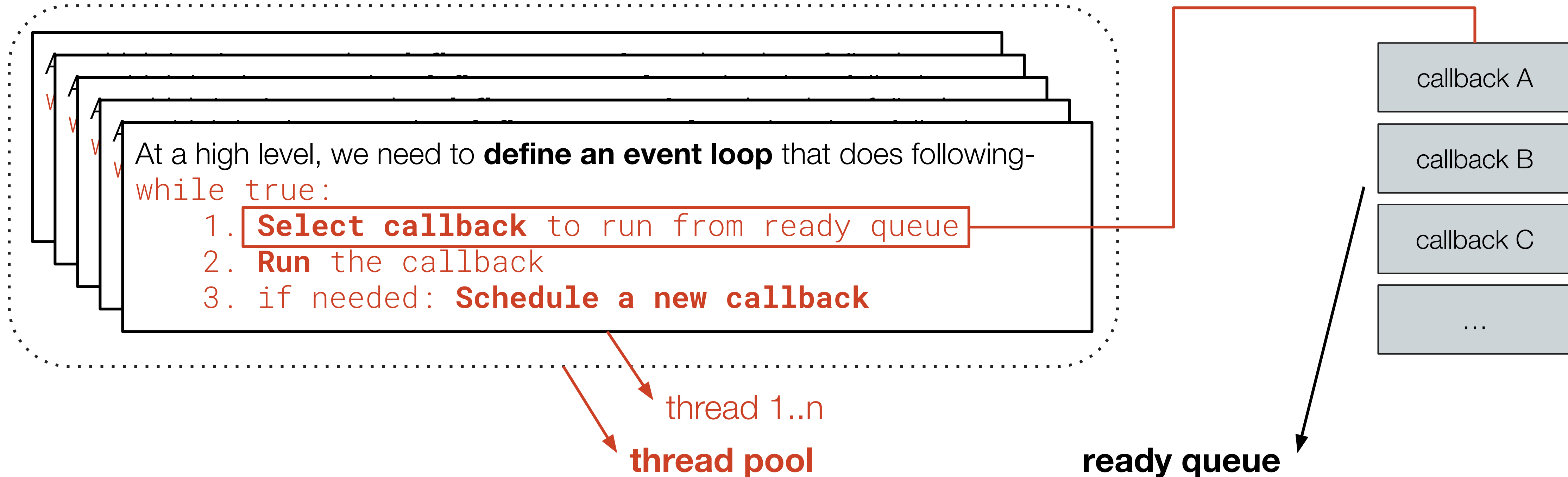
1. Start a task A, and do required computation
2. When task A blocks/waits, save state of A
3. Add check in selector to see if A is unblocked, if so it can be run again
4. Select another task to run and run it
5. Repeat for each task! (yep this is the loop)

At a high level, we need to **define an event loop** that does following-

```
while true:
   1. Select callback to run from ready queue
   2. Run the callback
   3. if needed: Schedule a new callback
```

Attempt 1: Event loop with a thread pool

1. The idea is to extend the **event loop for each thread** in a thread-pool.
2. We maintain a central ready queue, and each thread can pick from it



Attempt 1 Example: Old Go Runtime

The orignal Go runtime scheduler comprised of 3 major C `struct`

`struct G:`

Represented a **single goroutine**. Fields tracked **stack** and **current status**. It also contained **references to the code** that it is responsible for running.

`struct M:`

Go runtime's representation of an OS thread, had pointers to the **global queue of G's**, the **G that it is currently** running, its **own cache**, and a handle to **the scheduler**

`struct Sched:`

The Sched struct is a single, global struct that keeps track of the **different queues of G's and M's**, and more some information

Old Go Runtime: `struct G`

`struct G:`

Represented a **single goroutine**. Fields tracked **stack** and **current status**. It also contained **references to the code** that it is responsible for running.

```
struct  G
{
  byte*   stackguard; // stack guard information
  byte*   stackbase;  // base of stack
  byte*   stack0;     // current stack pointer
  byte*   entry;      // initial function
  void*   param;      // passed parameter on wakeup
  int16   status;     // status
  int32   goid;       // unique id
  M*      lockedm;    // used for locking M's and G's
  ...
};
```

Old Go Runtime: `struct M`

`struct M`:

Go runtime's representation of an OS thread, had pointers to the **global queue of G's**, the **G that it is currently** running, its **own cache**, and a handle to **the scheduler**

```
struct M
{
  G*        curg;               // current running goroutine
  int32     id;                 // unique id
  int32     locks;              // locks held by this M
  MCache    *mcache;            // cache for this thread
  G*        lockedg;            // used for locking M's and G's
  uintptr   createstack[32];    // Stack that created this thread
  M*        nextwaitm;          // next M waiting for lock
  ...
};
```

Old Go Runtime: `struct Sched`

`struct Sched:`

- There are **two queues containing G structs**, one is the runnable queue where M's can find work, and the other is a free list of G's.
- There is only **one queue pertaining to M's**; the M's in this queue are idle

```
struct Sched {
  Lock;              // global sched lock.
                     // must be held to edit G or M queues

  G *gfree;          // available g's (status == Gdead)
  G *ghead;          // g's waiting to run queue
  G *gtail;          // tail of g's waiting to run queue
  int32 gwait;       // number of g's waiting to run
  int32 gcount;      // number of g's that are alive
  int32 grunning;    // number of g's running on cpu
                     // or in syscall

  M *mhead;          // m's waiting for work
  int32 mwait;       // number of m's waiting for work
  int32 mcount;      // number of m's that have been created
  ...
};
```

In order to modify these queues, the **global Sched lock** must be held.

Old Go Runtime: Execution

1. The runtime starts out with several G's.
   One is in charge of **garbage collection**, another is in charge of **scheduling**, and one represents the **user's Go code**
2. Initially, one M is created to kick off the runtime.
   - As the program progresses, more G's may be created by the user's Go program, and **more M's may become necessary** to run all the G's
   - The runtime may provision additional threads up to `GOMAXPROCS`.
3. An M without a currently associated G will **pick up a G from the global runnable queue** and run the Go code belonging to that G.
4. If the Go code requires the M to block, for instance by invoking a system call, then another **M will be woken up** from the global queue of idle M's.

Attempt 1 Shortcomings

Single global mutex: `Sched.Lock`:

The shared ready queue will need a lock, which becomes a bottleneck at high scales (~100k callbacks/lightweight threads)

Vitess (implemented in Golang) server would max out at 70% CPU, and spend 14% of all time in `runtime.futex()`

Attempt 1 Shortcomings

Single global mutex: `Sched.Lock`:

The shared ready queue will need a lock, which becomes a bottleneck at high scales (~100k callbacks/lightweight threads)

Memory resources: Per-M memory cache (`M.mcache`):

Memory resources owned at thread level, are in use *even when the thread is not running anything* (which is often).

Memory cache and other caches (stack alloc) are associated with all M's, and not with M's running Go code. **Ratio between M's running Go code and all M's can be as high as 1:100**

Attempt 1 Shortcomings

Single global mutex: `Sched.Lock`:

The shared ready queue will need a lock, which becomes a bottleneck at high scales (~100k callbacks/lightweight threads)

Memory resources: Per-M memory cache (`M.mcache`):

Memory resources owned at thread level, are in use even when the thread is not running anything (which is often). A ratio between M's running Go code and all M's can be as high as 1:100.

Aggressive thread blocking/unblocking:

In presence of syscalls worker threads are frequently blocked and unblocked. This adds a lot of overhead.

Attempt 2

Work stealing scheduler

Attempt 2: Work stealing scheduler

1. The core idea is to remove the **central ready queue**, and have a queue for each process, this solves lock bottleneck

2. To distribute work in a **decentralised way**, concept of “stealing”, work can be used (Note, similar concept of “sharing” exists too)

Every task is created on some thread, and is associated with it. When a thread is idle **(has no tasks to run) it can steal “work”** (tasks) from other threads!

3. We also introduce **indirection in resource ownership**, to handle other bottlenecks

Attempt 2 Example: Work stealing scheduler in Go

Include another `struct`, P, to simulate processors. There are exactly `GOMAXPROCS` P's, and a P would be another required resource for an M in order for that M to execute Go code.

An M would still represent an OS thread, and a G would still portray a goroutine.

```
struct P
{
	Lock;
	G *gfree; // freelist, moved from sched
	G *ghead; // runnable, moved from sched
	G *gtail;
	MCache *mcache; // moved from M
	FixAlloc *stackalloc; // moved from M
	uint64 ncgocall;
	GCStats gcstats;
	// etc
	...
};

P *allp; // [GOMAXPROCS]
```

Work stealing scheduling steps

1. When an M is willing to start executing Go code, it must **pop a P form a free list of P**. When an M ends executing Go code, it pushes the P to the list.
2. When a new G is created or an existing G becomes runnable, it is **pushed onto a list of runnable goroutines** of current P.
3. When P finishes executing G, it first **tries to pop a G from own list** of runnable goroutines.
4. If the list is empty, **P chooses a random victim** (another P) and tries to **steal a half of runnable goroutines** from it.

Resolving other bottlenecks

- Threads (M) acquire resources when they are running a go-routine (G), this indirection cuts memory waste
- Note, OS does not know about P, as it is an abstraction, but it is aware of M, hence affinity between M, and G leads to better locality
- When an M creates a new G, it must ensure that there is another M to execute the G (if not all M's are already busy). Similarly, when an M enters syscall, it must ensure that there is another M to execute Go code. This is done with (a mix of passive i.e. `sched_yield` and active spinning)

Scalable Go Scheduler Design Doc

Conclusion

Comparing Goroutines and OS threads

Goroutines blocking

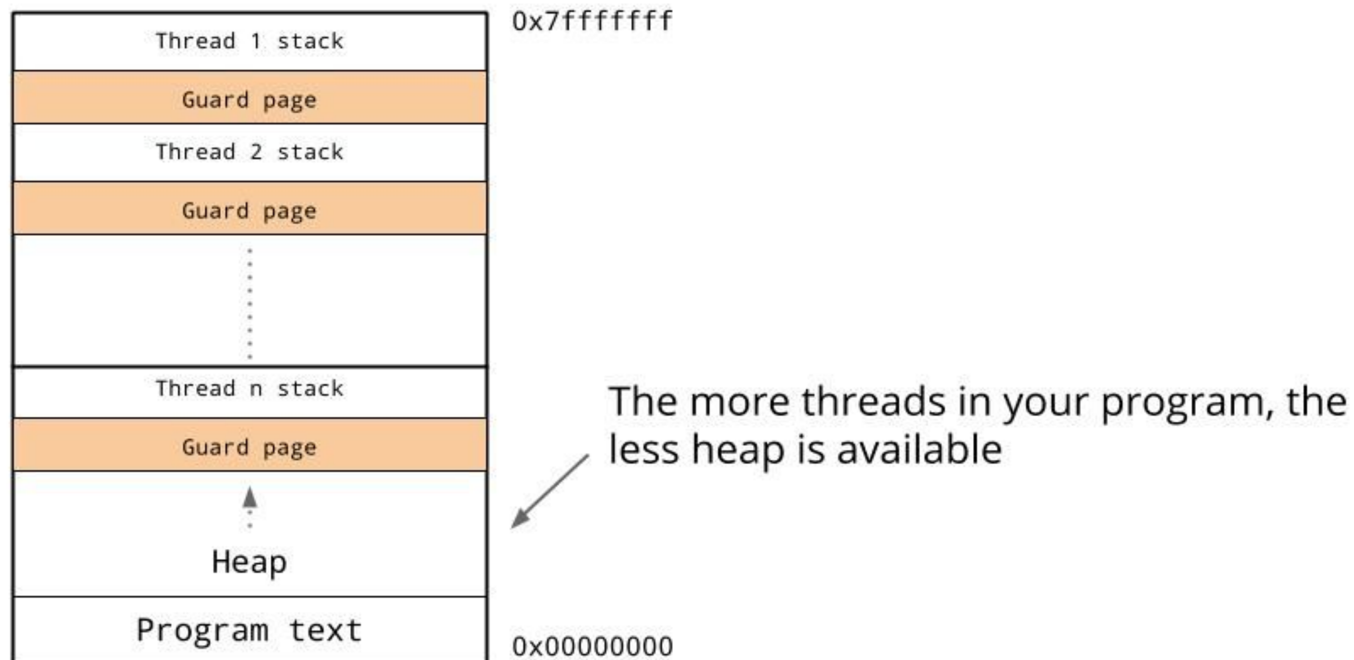
Goroutines are cheap and do not cause the thread on which they are multiplexed to block if they are blocked on

1. **network input**
2. **sleeping**
3. **channel operations**
4. **primitives in the sync package.**

Stack memory requirement

OS Thread

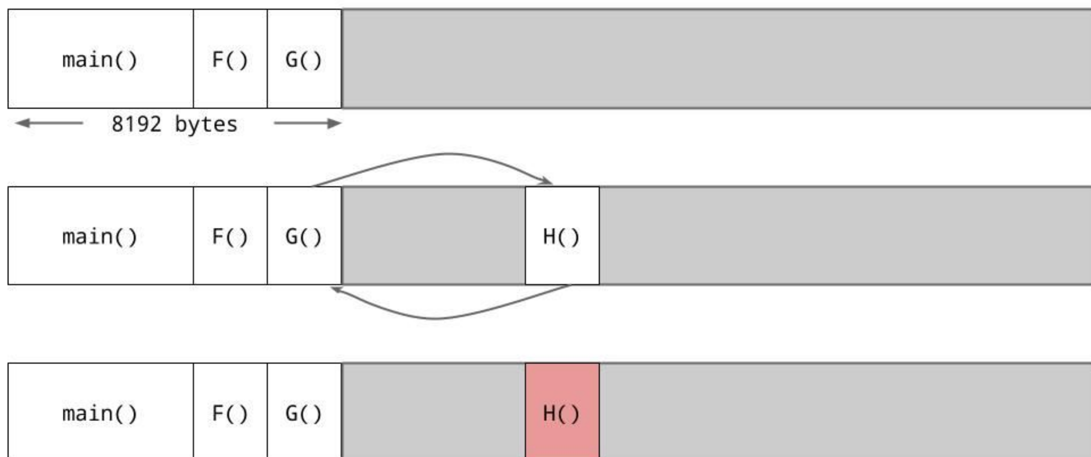
OS threads have allocated stacks with size(s) in MBs, and each thread has a guard page between their stacks. This starts to matter at scale.

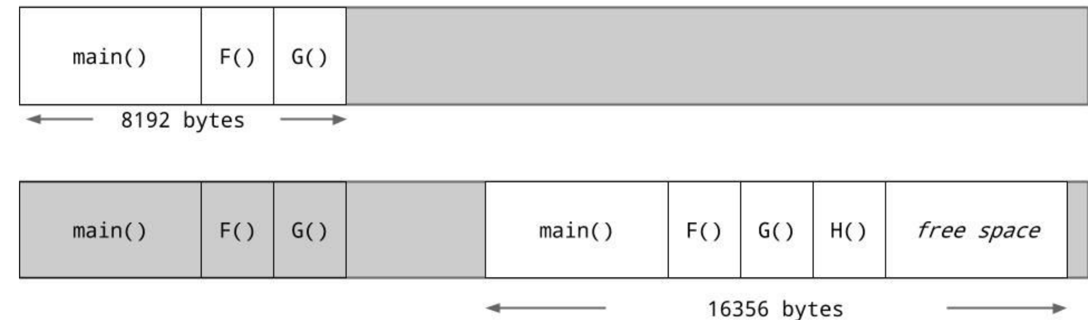


Stack memory requirement

Goroutines Goroutine stacks are not memory intensive (measured in KBs). Instead of using guard pages, the Go compiler **inserts a check** as part of every function call to check if there is sufficient stack for the function to run.

If there is not, the **runtime can allocate more stack space** (Go copies the existing stack into allocated memory, can you guess why?)





How Stacks are Handled in Go.

Thread switching cost

OS Thread

Threads are scheduled preemptively. And context switches are very expensive!

Goroutines

Goroutines are scheduled cooperatively and when a switch occurs, only 3 fields need to be saved/restored - Program Counter, Stack Pointer and DX

Massively concurrent IO

OS Thread (i.e. using OS threads directly for concurrency)

Modern networking primitives like `io_uring`, and kernel bypass (which we will cover soon) are needed to scale up

Goroutines

Goroutines easily handle O(~100k) concurrent IO requests, note, using (1) no thread when blocked, (2) the low memory requirements make this possible