The C∀ Scheduler

PhD Comprehensive II Research Proposal

Thierry Delisle

September 1, 2020

C∀ and Concurrency

Concurrency in C∀

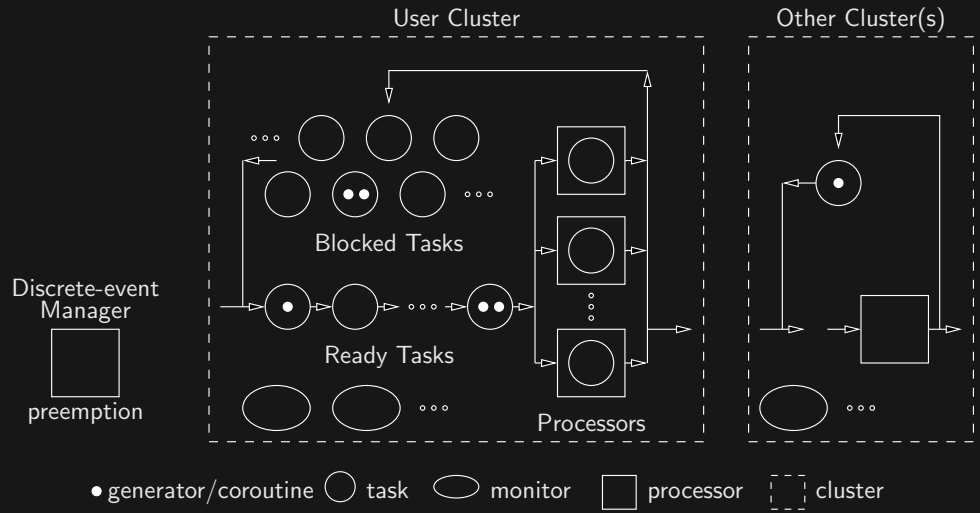
User Level Threading

- M:N threading.
- User Level Context Switching causes kernel-threads to run a different user-thread.

Threads organized in clusters:

- Clusters have their own kernel threads.
- Threads in a cluster are on run on the kernel threads of that cluster.

Concurrency in C∀



Scheduling goal for C∀

The C∀ scheduler should be *viable* for any workload.

This implies:

1. Producing a scheduler with sufficient fairness guarantees.
2. Handling kernel-threads running out of work.
3. Handling blocking I/O operations.

Scheduling in Practice

In the Wild

Schedulers found in production application generally fall into two categories:

- Feedback Scheduling
- Priority Scheduling (explicit or not)

Feedback Scheduling

Most operating systems based their scheduling on feedback loops.

The scheduler runs a thread and adjusts some metric to choose when to run it, e.g., least CPU time first.

Relies on the following assumptions:

1. Threads live long enough for useful feedback information to be to gathered.
2. Threads belong to multiple users so fairness across threads is insufficient.

Priority Scheduling

Runs all ready threads in group A before any ready threads in group B.

Explicit priorities:

- Threads given a priority at creation, e.g., Thread A has priority 1, Thread B has priority 6.

Implicit priorities:

- Certain threads are preferred, based on various metrics, e.g., last run, last run on this CPU.

Priority Scheduling: Work-Stealing

Work-Stealing is a very popular strategy.

Algorithm

1. Each processor has a list of ready threads.
2. Each processor runs threads from its ready queue first.
3. If a processor's ready queue is empty, attempt to run threads from some other processor's ready queue.

Work-Stealing has implicit priorities: For a given processor, threads on it's queue have higher priority.
Processors begin busy for long periods can mean starvation.

Project: Proposal & Details

Central Ready-Queue

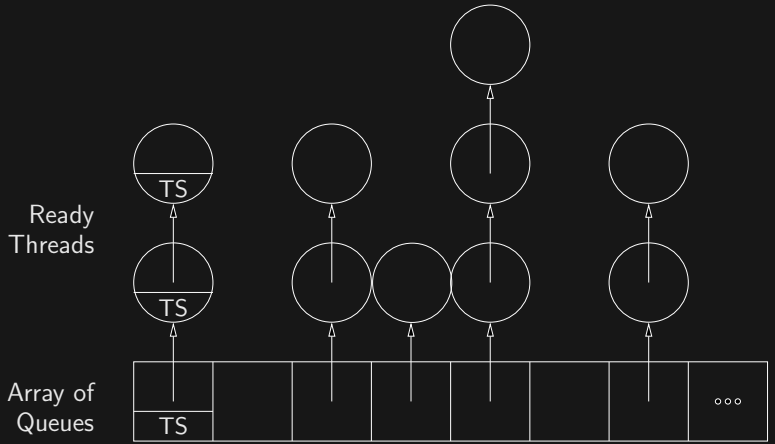
C∀ will have a single ready-queue per cluster.

The ready-queue will be sharded internally to reduce contention.

No strong coupling between internal queues and processors.

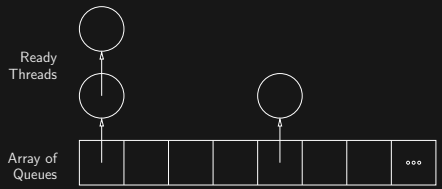
Constrasts with work-stealing which has a queue per processor.

Central Ready-Queue



Central Ready-Queue Challenges

Semi-"Empty" ready-queues means success rate of randomly guessing goes down.



Possible solutions:

- Data structure tracking the work, can be dense or sparse, global or sharded.
- Add bias towards certain sub-queues.

Dynamic Resizing

Processors can be added at anytime on a cluster.

The array of queues needs to be adjusted in consequence.

Solution: Global Reader-Writer lock

- Acquire for reading for normal scheduling operations.
- Acquire for right when resizing the array and creating/deleting internal queues.

Idle Sleep

Processors which cannot find threads to run should sleep, using `pthread_cond_wait`, `sigwaitinfo`, etc.

Scheduling a thread may *need* to wake sleeping processors.

- Threads can be scheduled from processors terminating or running outside the cluster. In this case, all processors on the cluster could be sleeping.

If *some* processors are sleeping, waking more may be wasteful. A heuristic for this case is outside the scope of this project.

Asynchronous I/O

- I/O Operations should block user-threads rather than kernel-threads.

- This requires 3 components:
  1. an OS abstraction layer over the asynchronous interface,
  2. an event-engine to (de)multiplex the operations,
  3. and a synchronous interface for users to use.

Asynchronous I/O: OS Abstraction

`select`

"select() allows a program to monitor multiple file descriptors, waiting until one or more of the file descriptors become "ready" for some class of I/O operation."

— Linux Programmer's Manual

+ moderate overhead per `syscall`
- Relies on `syscalls` returning `EWOULDBLOCK`.

Asynchronous I/O: OS Abstraction

epoll

More recent system call with a similar purpose.

+ Smaller overhead per `syscall`.
+ Shown to work well for sockets.
- Still relies on `syscalls` returning `EWOULDBLOCK`.
- Does not support linux pipes and TTYs.

Asynchronous I/O: OS Abstraction

Kernel Threads

Use a pool of kernel-threads, to which blocking calls are delegated.

- ▶ Technique used by many existing systems, e.g., Go, libuv
- \+ Definitely works for all `syscalls`.
- − Can require many kernel threads.

Asynchronous I/O: OS Abstraction

io_uring

A very recent framework for asynchronous operations available in Linux 5.1 and later. Uses two ring buffers to submit operations and poll completions.

+ Handles many `syscalls`.
+ Does *not* rely on `syscalls` returning `EWOULDBLOCK`.
− Requires synchronization on submission.
− System call itself is serialized in the kernel.

Asynchronous I/O: Event Engine

An event engine must be built to fit the chosen OS Abstraction.

The engine must park user-threads until operation is completed.

Depending on the chosen abstraction the engine may need to serialize operation submission.

Throughput and latency are important metrics.

Asynchronous I/O: The interface

The Asynchronous I/O needs an interface.

Several options to take into consideration:

- Adding to existing call interface, e.g., `read` and `cfaread`.
- Replacing existing call interface.
- True asynchronous interface, e.g., callbacks, futures.

Conclusion

Summary

Runtime system and scheduling are still open topics.

This work offers a novel runtime and scheduling package.

Existing work only offers fragments that users must assemble themselves when possible.

Timeline

May 2020 Oct 2020	Creation of the performance benchmark.
Nov 2020 Mar 2021	Completion of the implementation.
Mar 2021 Apr 2021	Final performance experiments.
May 2021 Aug 2021	Thesis writing and defense.

Timeline

Questions?