

COOPERATIVE TASK MANAGEMENT WITHOUT MANUAL STACK MANAGEMENT.

By Adya et al., Microsoft Research.

USENIX Annual Technical Conference 2002

This paper in a nutshell:

- Two programming paradigms for concurrency
 - ↳ Threads
 - ↳ Event-driven programming.
- In threads, programmer focuses on task at hand, leaving decision on where to yield thread to underlying runtime.
- In event-driven programming, programmer breaks a task down into "event handlers". Event handler boundaries are yield points.
- In threads, no control over yielding.
- In events, task logic "broken up" into events.
- THIS PAPER: Can get the best of both worlds.
NEW IDEA: STACK RIPPING. ~~STACK MANAGEMENT~~
CAN COEXIST WITH THREADS

Let us first define 5 concepts related to managing concurrency (not entirely unrelated).

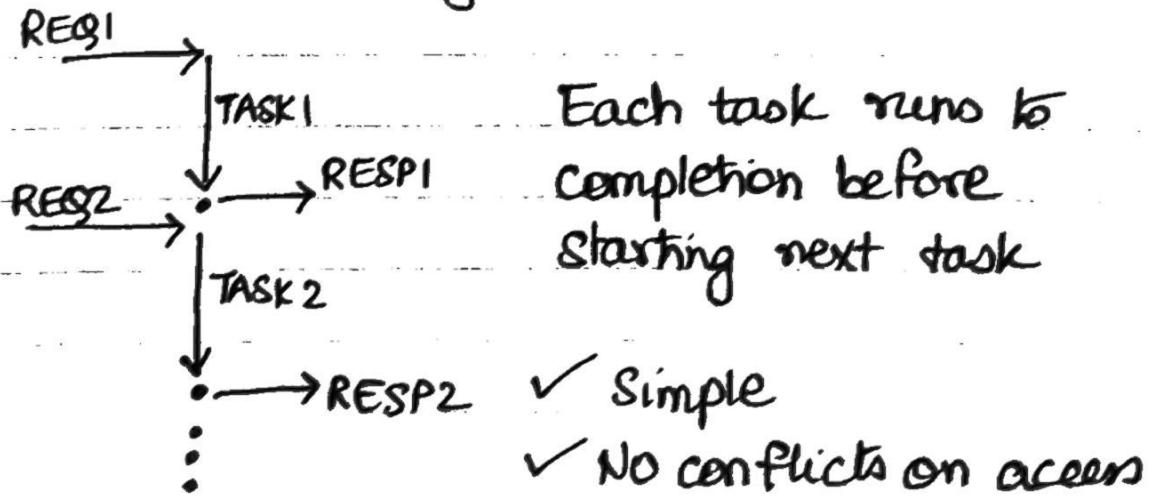
CONCEPT 1: TASK MANAGEMENT.

What is a "task"? A high-level objective that a program accomplishes.

E.g., A Web server serving a page: Task consists of (1) receiving a network request (2) reading a block of data from disk & (3) serving that data in response to the request.

How to manage tasks? Three possible options:

(a) Serial task management:



Each task runs to completion before starting next task

✓ Simple

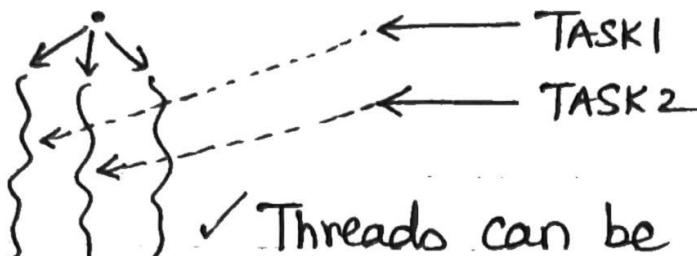
✓ No conflicts on access

to shared state (e.g. web page counter)

✗ Does not fit well with multiprocessor parallelism.

(b) PRE-EMPTIVE TASK MANAGEMENT.

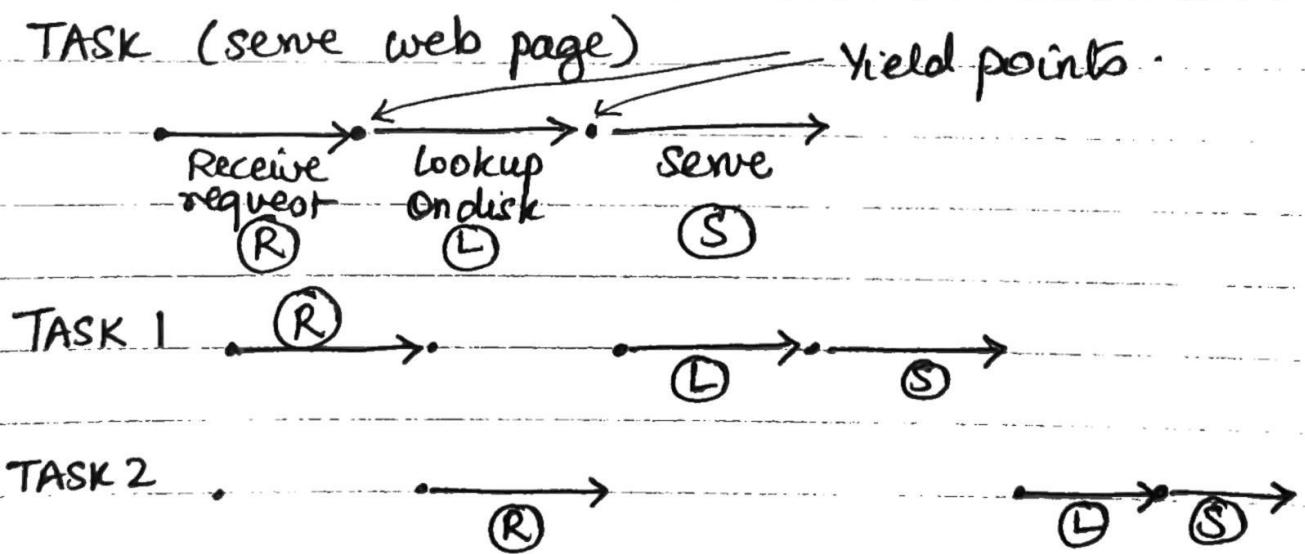
a.k.a threaded programming



✓ Threads can be pre-empted and interleave arbitrarily on a uniprocessor (an overlap on a multiprocessor).

(c) Cooperative task management.

- A task yields control to other tasks only at well-defined points in its execution.



(on a uniprocessor, for example).

Each of these steps is called an event handler

EACH EVENT HANDLER cannot be pre-empted.

What is the advantage of serial task management?

→ Easy to reason about global invariants

→ Web page hit counter

$$\left\{ \begin{array}{l} x = \text{counter} \\ x = x + 1 \\ \text{counter} = x \end{array} \right\} \text{ guaranteed to execute atomically.}$$

→ Not so in pre-emptive task management. So programmer must use synch. primitives:

COOPERATIVE THREAD MANAGEMENT preserves

some of this advantage

- Event handlers are atomic
- So any invariants on global state can be expected to hold on task boundaries
(e.g., page counter strictly increases on access)

BUT I/O may have to become asynchronous.

More soon on that.

CONCEPT 2 : STACK MANAGEMENT.

What "stack" are we referring to?

The runtime stack of activation records.

As a task runs, it may need to call various functions. These functions are part of the stack in a caller → callee relationship.

Task (~~serve~~ ^{Serve} Page)

Thread

```
serve page () {
    // fetch net req()
    // read from disk()
    // serve()
}
```

Event handlers

Each of the
fetchnet req(),
read from disk()
and serve()
is an event handler.

As thread executes,
programmer just needs to
focus on task at hand.

Programmer must explicitly
manage invariants are
held at event handler
boundaries.

No need to worry about
thread stack state if
thread is pre-empted

Threaded approach called
"automatic stack management"
Since the language itself takes care of it.

Event-based approach called
"manual stack management"
Since programmer has identified the yield
points, and knows the possible stack configs
on a yield.

CONCEPT 3: I/O MANAGEMENT.

- ↳ Synchronous, or, blocking I/O
- ↳ Asynchronous, or, non-blocking I/O.

Asynchronous I/O ops can overlap.

NOTE:

Concurrency in I/O is different from
concurrency in tasks management because
I/O does not use CPU & it does not
access state of computation.

CONCEPT 4: CONFLICT MANAGEMENT

- Need to ensure atomicity.
- Fine-grained locking vs. coarse-grained locks.
- Pessimistic approach (lock out others)
vs.

Optimistic approach (speculatively execute & then rollback on conflict)

Event-based programming naturally decomposes the tasks into units that are atomic.

All locks, etc., released at event handler boundaries.

CONCEPT 5: DATA PARTITIONING

- Partition data (global state) into different units to reduce opportunities for conflict.

Our primary focus in this lecture

→ Task management
&

→ Stack management.

Let's examine STACK MANAGEMENT in detail.

Automatic stack management

(Top-level task expressed as a single logical procedure at language level)
 (that procedure can call others, that is okay).

```
serve page() {
    int page_counter;
    // fetch request
    ... (network I/O)
    // obtain page from disk
    ... (disk I/O)
    // serve page to user
    ... (computation + network I/O)
```

}

- I/O operations could block
- "State" (page counter) is part of the language stack (e.g. stack of the thread).

VS.
 event-driven style

mainevent loop() {
 ↳ dispatch switch case
 ↳ fetch net request()
 ↳ read from disk()
 ↳ render page()
}

fetch net request() {
 ... // Network I/O : even these can be asynchronous.

Raise netrequestdone

}

read from disk() { // invoked when netrequestdone is raised
 ... // Disk I/O counter modified here.

Raise readfromdiskdone

}

renderpage() { // invoked when readfromdisk is done.
 ... // Net I/O

}

- Event handlers run to completion
- Need to register "continuations" for the part of the logic that must run later.

EXAMPLE:

A K/V store on disk with an in-memory cache

Automatic Stack Mgmt

```
VALUEOBJ GetValue (KEY) {
    VALUE = LookupMemKVStore (key);
    if (VALUE ≠ NULL) return VALUE;
    VALUE = new VALUEOBJ();
    DISKREAD (KEY, &VALUE)
    Insert MemKVStore (KEY, VALUE);
    return VALUE;
}
```

This is the blocking approach. Let's look at manual stack management. The diskread is made asynchronous:

```
VALUEOBJ GetValue 1 (KEY) {
    VALUE = LookupMemKVStore (key);
    if (VALUE ≠ NULL) return value;
    VALUE = new VALUEOBJ();
    CONTINUATION * cont =
        new CONTINUATION (GetValue 2, key, value)
    EVENT HANDLER eh = AsynchRead (KEY, VALUE);
    REG HANDLER (eh, cont);
```

3.

GetValue2() {

} → the one that ~~looks~~ inserts the returned value into the in-mem hash table.

Essentially, the programmer now responsible for keeping track of what computation is pending, and package that up as a "continuation".

Programming style goes from thinking about the logical flow of the task to that of events and continuations.

The I/O call in between that we wanted to handle with events requires us to "rip apart" the logic into 2 functions. (Same with the web server example).

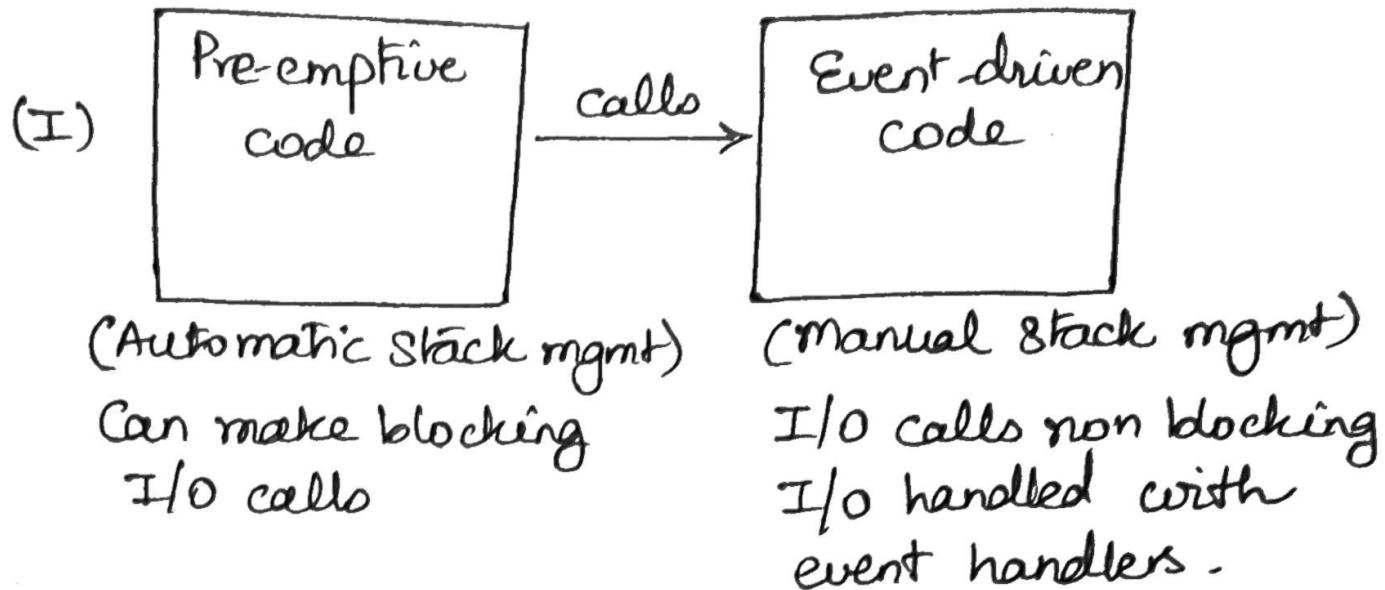
This is called "stack ripping".

This paper is in the context of windows.

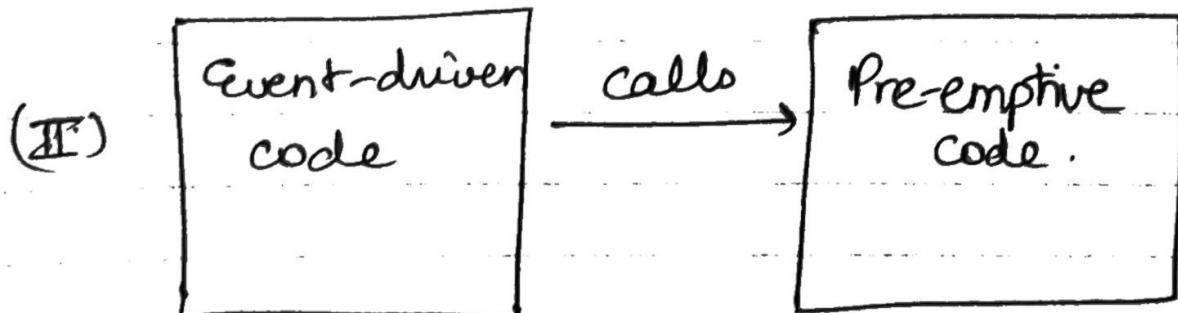
- Threads → preemptive
- Fibres → suited for cooperative task mgmt

Main contribution: Design patterns for both those kinds of code to work well together.

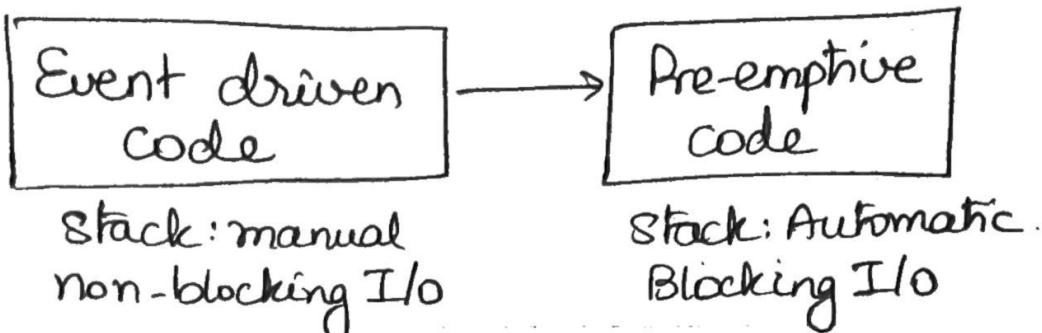
There are two cases.



and



Let's consider Case II first



What is the problem?

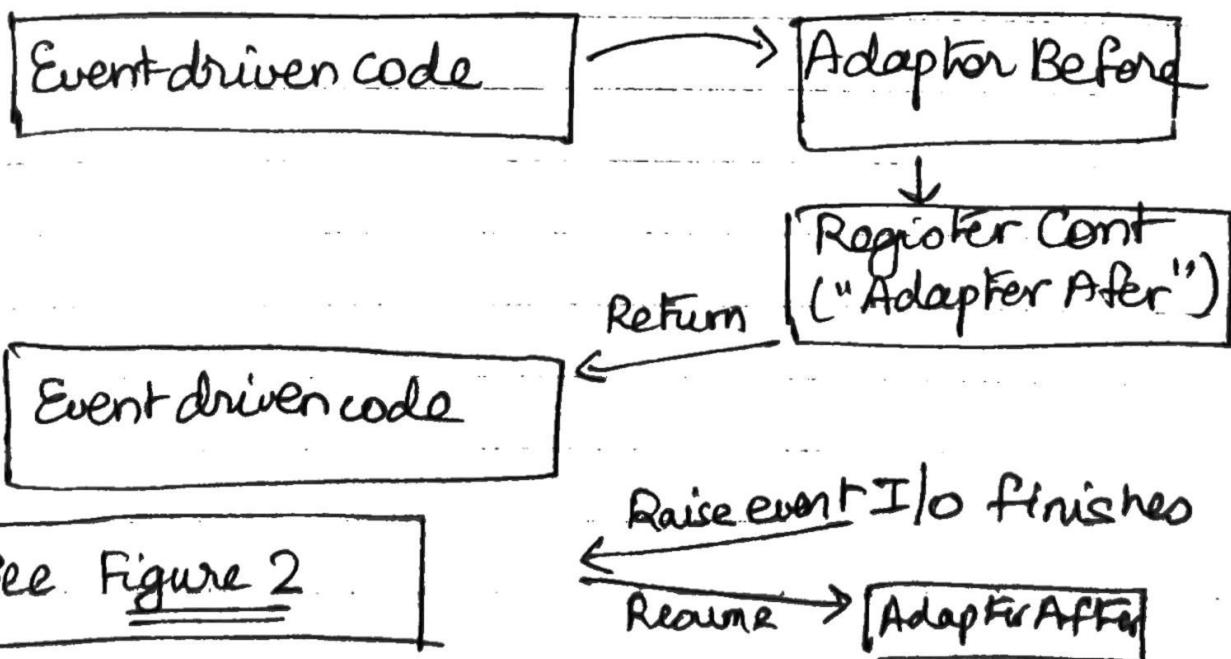
the event code calls into something that can block potentially.

Idea: don't call directly.

use an adaptor instead

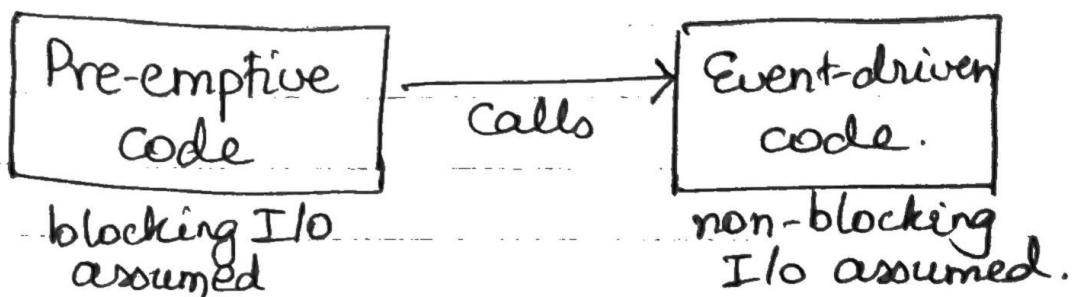
Adaptor splits pre-emptive code logically, into 2 parts : "before I/O", "after I/O"

"After I/O" code ~~opti~~ packaged as a continuation that is called later



The adaptor can be automatically generated and does the task of "stack ripping" ["CFA Adaptor" in paper]

Now consider Case I



Again, the idea is to use an adaptor. Adaptor is called by the pre-emptive code & "blocks". It calls the event handler, and has the continuation that handles the event on I/O completion. The code then unblocks and returns to pre-emptive code.

This is in Figure 3 of the paper

