EC-ENG 231 (Spring 2024)

Software Organization: CPU Sharing

Fatima Anwar

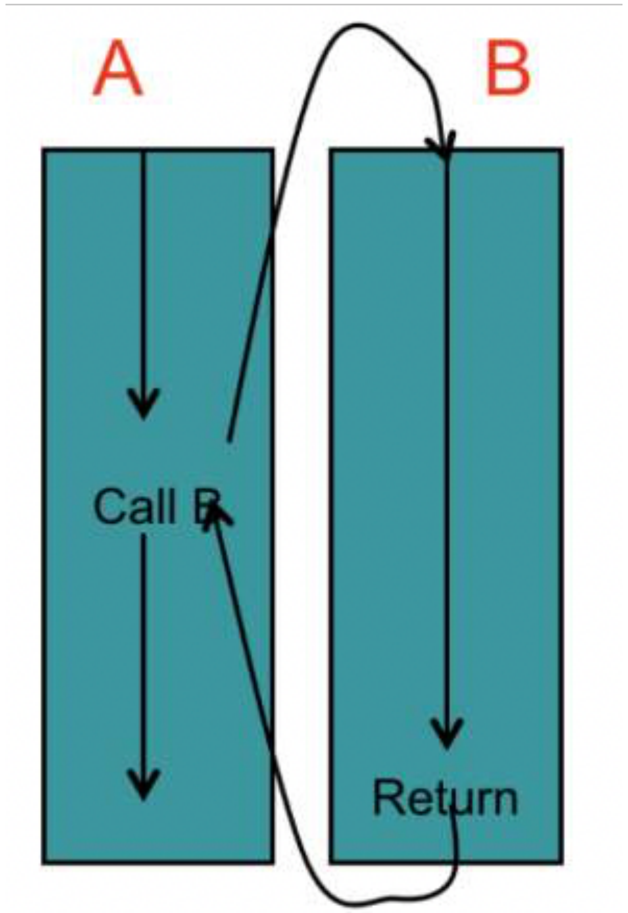
fanwar@umass.edu

UMass Amherst

1. Subroutines

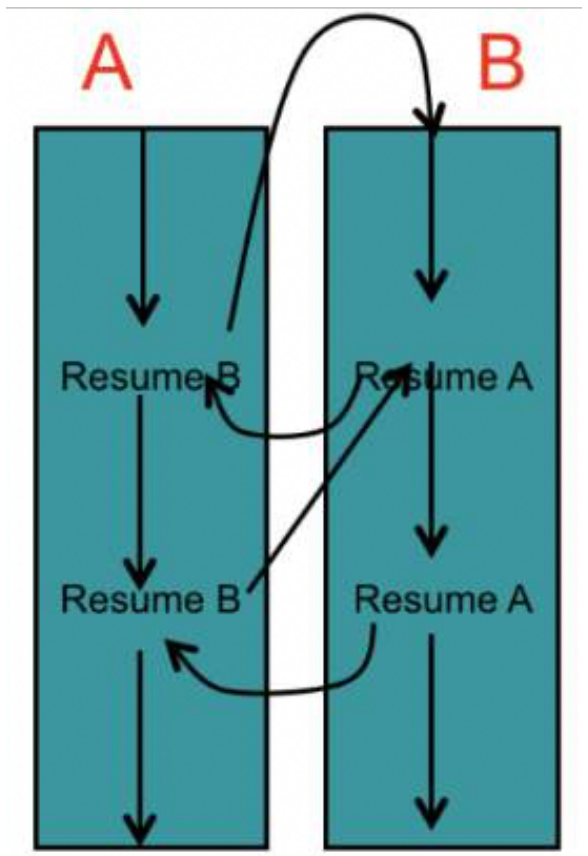
- A piece of program (B) that can be called during the execution of another program (A)

- function in C is the most common example of subroutines



2. Coroutines

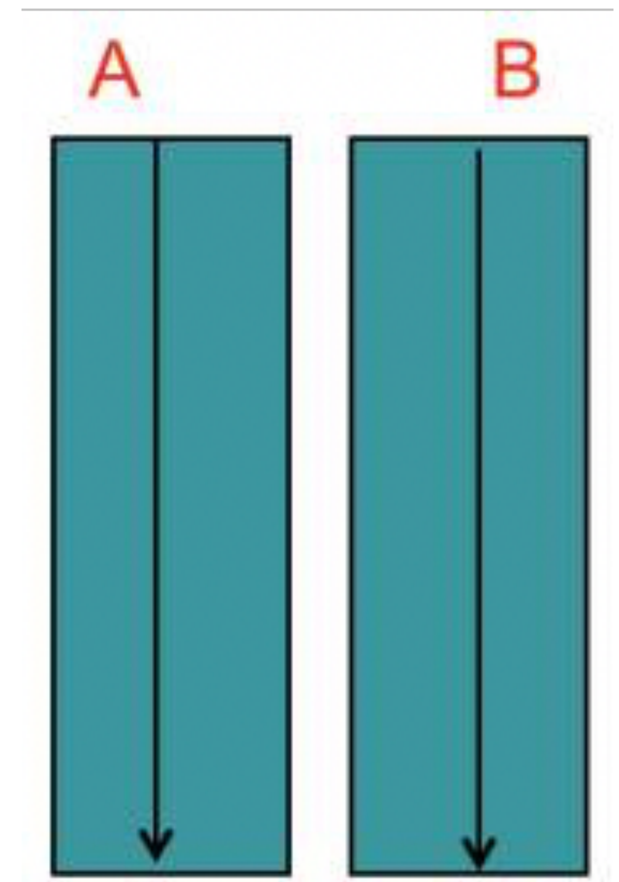
- Two or multiple programs taking turns to use CPU
- As one program runs, the others are suspended
  - Programs voluntarily yield control either periodically or when idle/ logically blocked
- Type of a **cooperative multitasking** or non-preemptive multitasking
- No OS level **context switch**



In cooperative multitasking, all tasks must collaborate. If one doesn't cooperate, it may use all of the processor resources

3. Threads

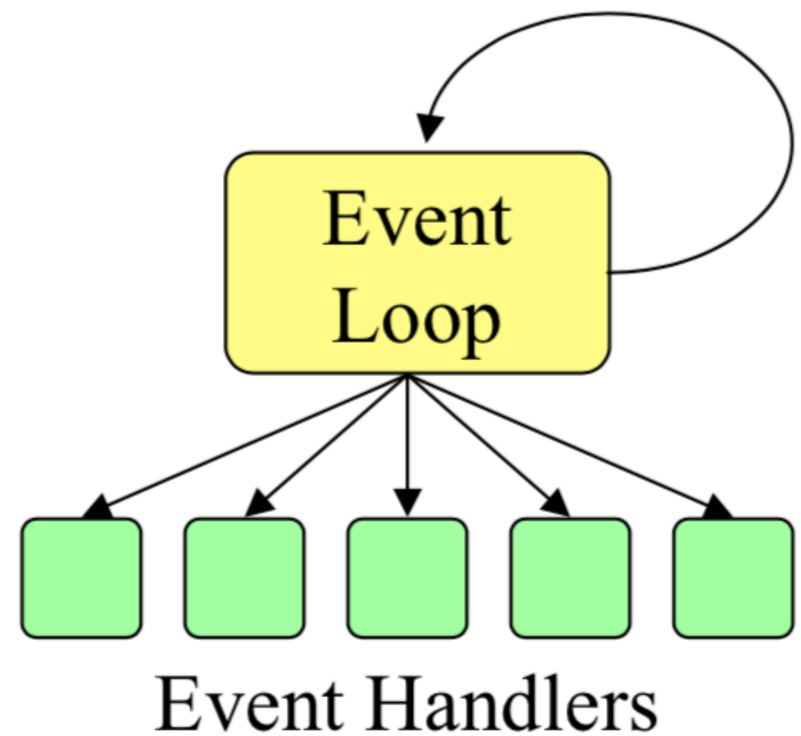
- Thread provides developers with a useful abstraction of concurrent execution
- Threads enable **preemptive multitasking**
- Examples
  - Interrupts suspend ongoing process and processor runs ISR
  - OS initiates a context switch between processes based on pre-defined priorities



Preemptive multitasking forces tasks to share the processor, whether they want to or not

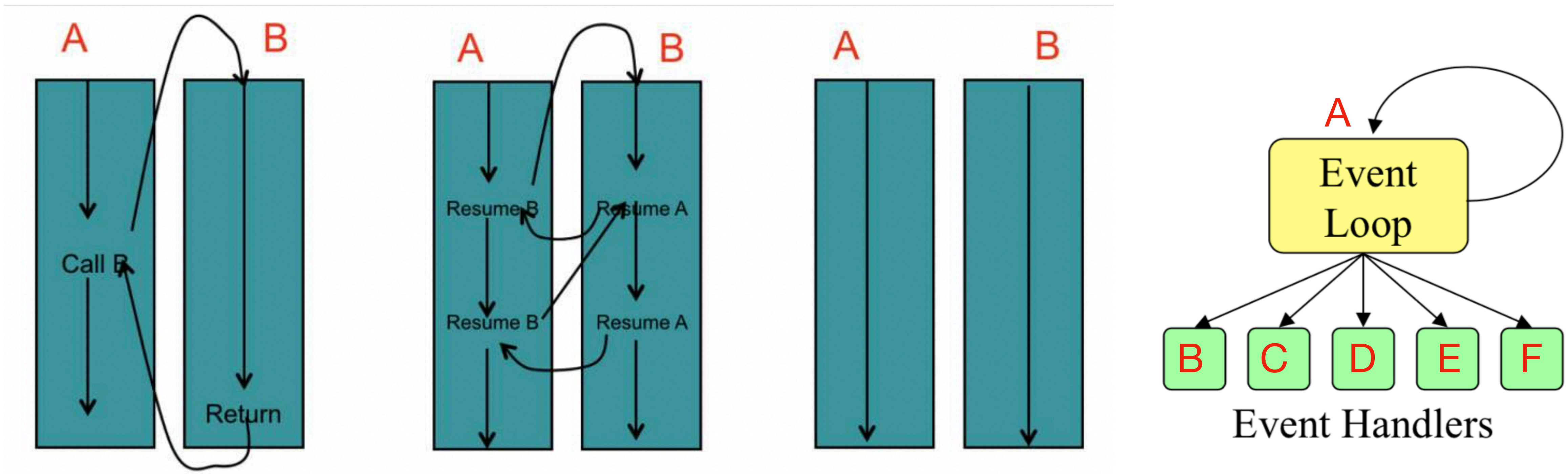
4. Events

- One execution stream
- Register interest in events
- Event loop waits for events, invokes handlers
- No preemption of event handlers (callbacks)
- Handlers generally short-lived
- Used for:
  - GUIs:
    - One handler for each event (press button, invoke menu entry etc.)
  - Embedded systems, distributed systems, web servers
    - One handler for each source of input
    - Handler processes requests
    - Event-driven I/O for I/O overlap



Wait for events, and execute them to completion

CPU Sharing Techniques



SUBROUTINES
Hierarchical
Sequential

COROUTINES
Symmetric
Sequential

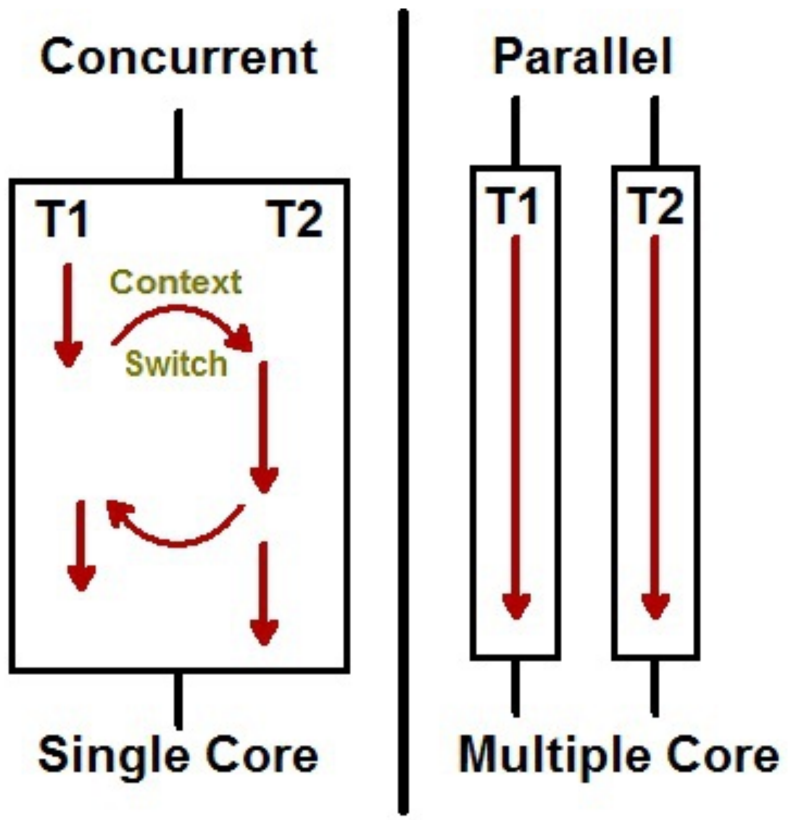
THREADS (PROCESSES)
Symmetric
Concurrent

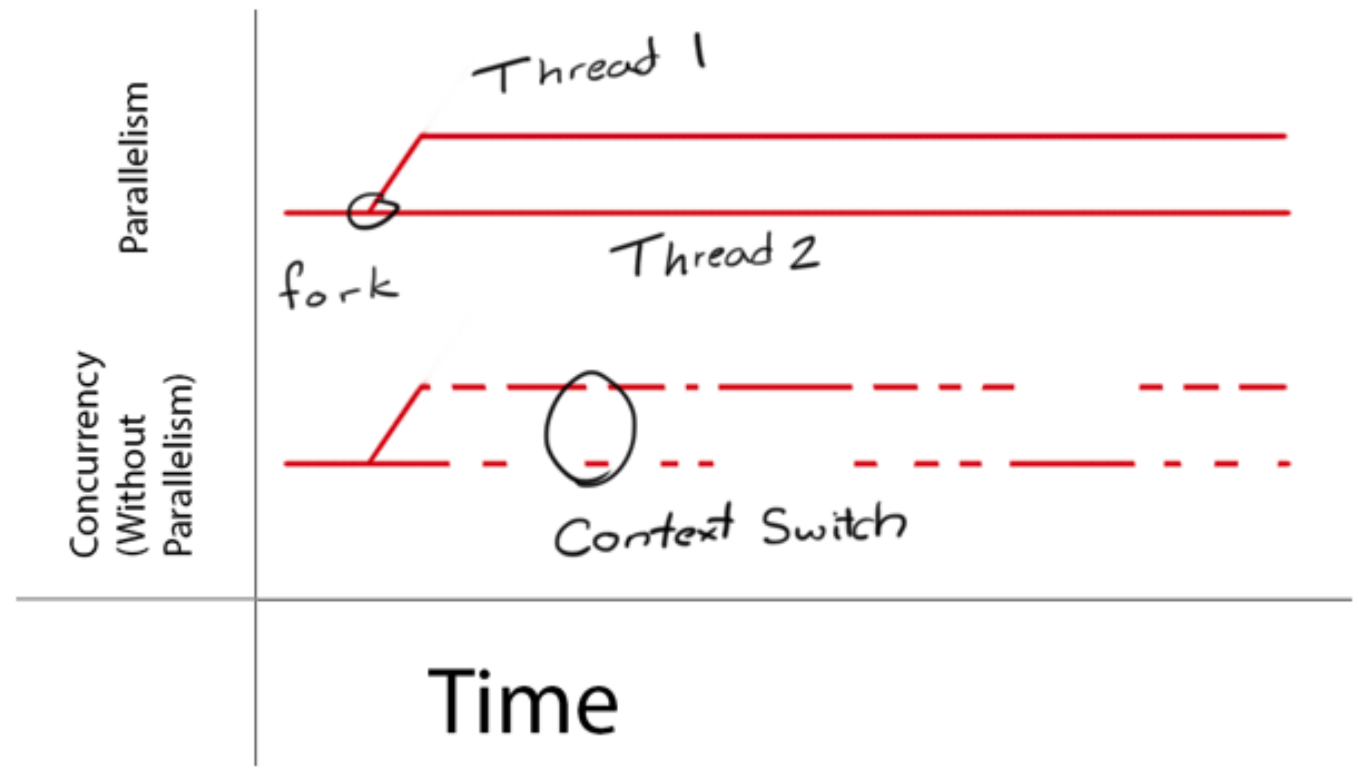
EVENTS
Condition-based
Sequential

Software Organization: Concurrency

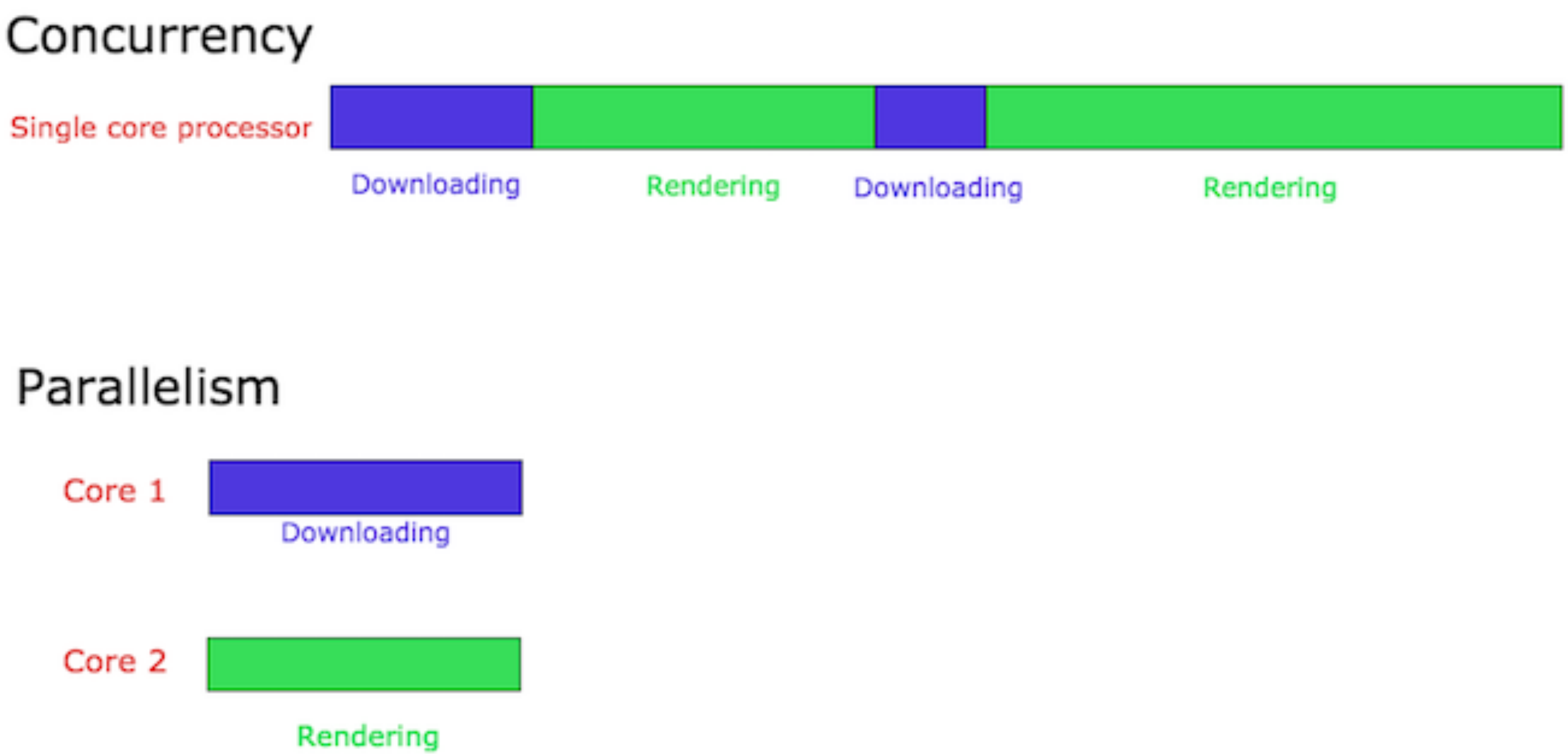
- **Concurrency** means multiple computations are happening at the same time.
  - It is the ability to execute multiple tasks out-of-order or in partial order, without affecting the final outcome
- Embedded systems receive many events from the outside
- Must be able to handle unknown order, arbitrary times, and in parallel
  - Managing concurrency is the key!
- **Multi-Threaded** v/s **Event-Driven**

Concurrency v/s Parallelism





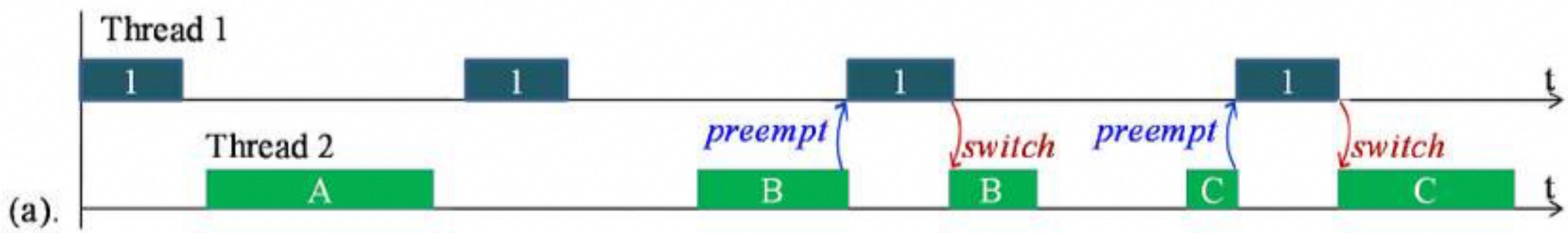
Concurrency v/s Parallelism: Example



Concurrency gives an ***illusion*** of parallelism

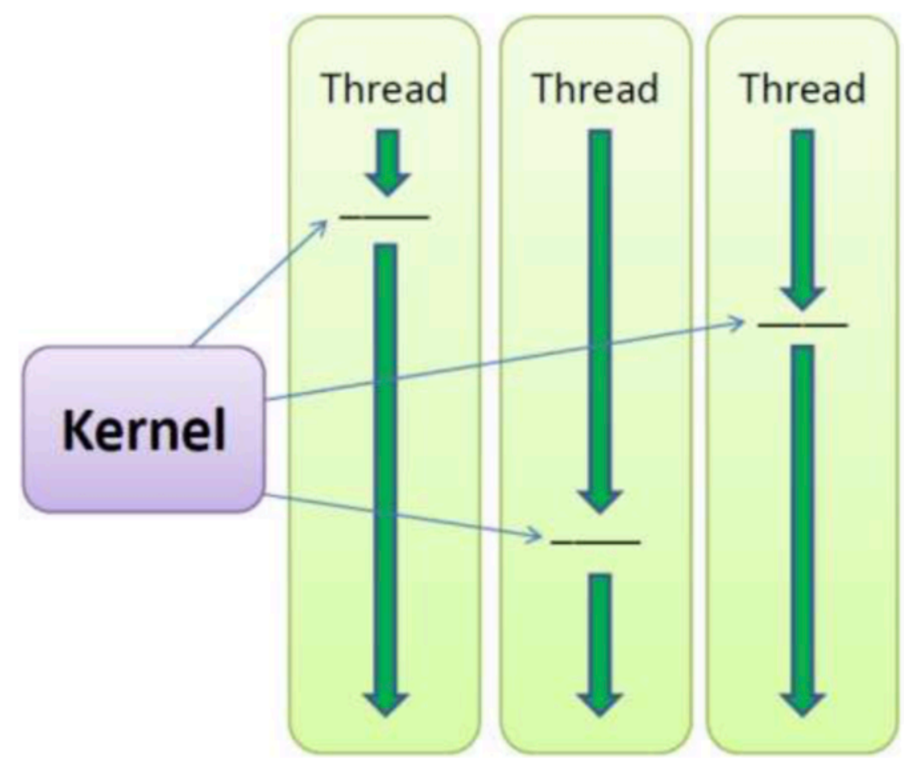
1. Multi-Threaded

- Threads can preempt each other, each maintaining an individual run-time stack



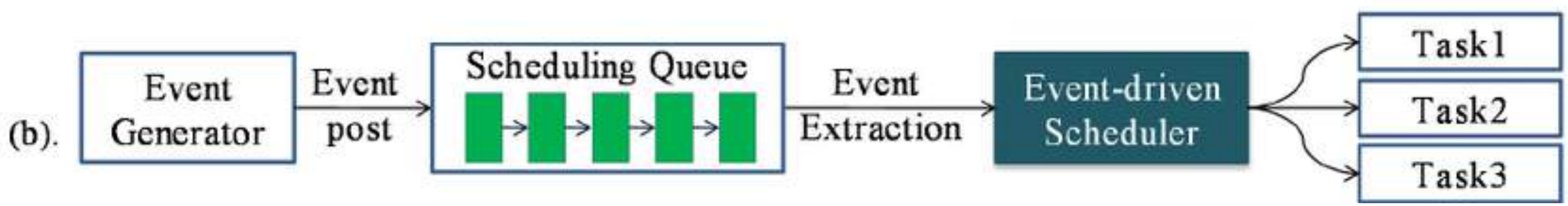
- Software organized as multiple threads each with sequential code

- Processor switches between them as needed

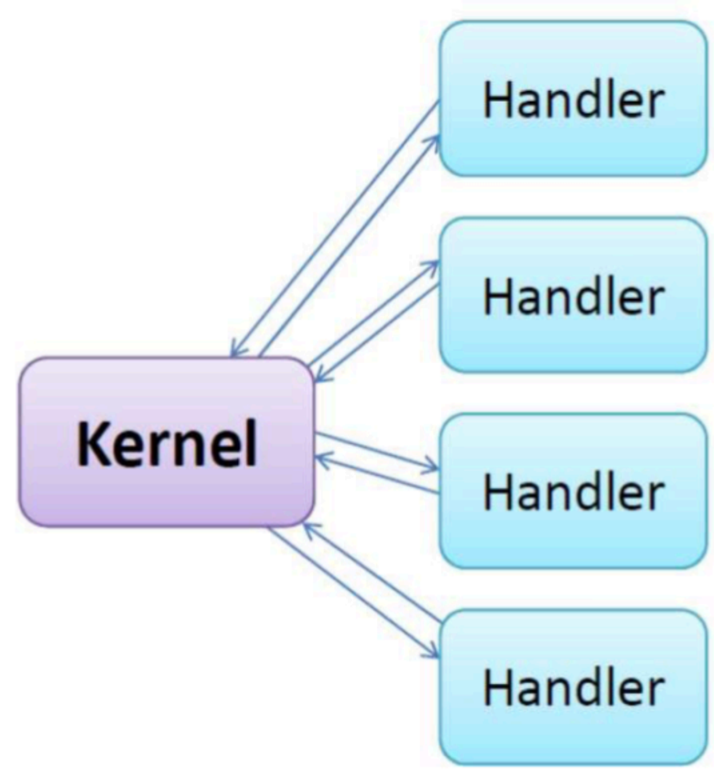


2. Event-Driven

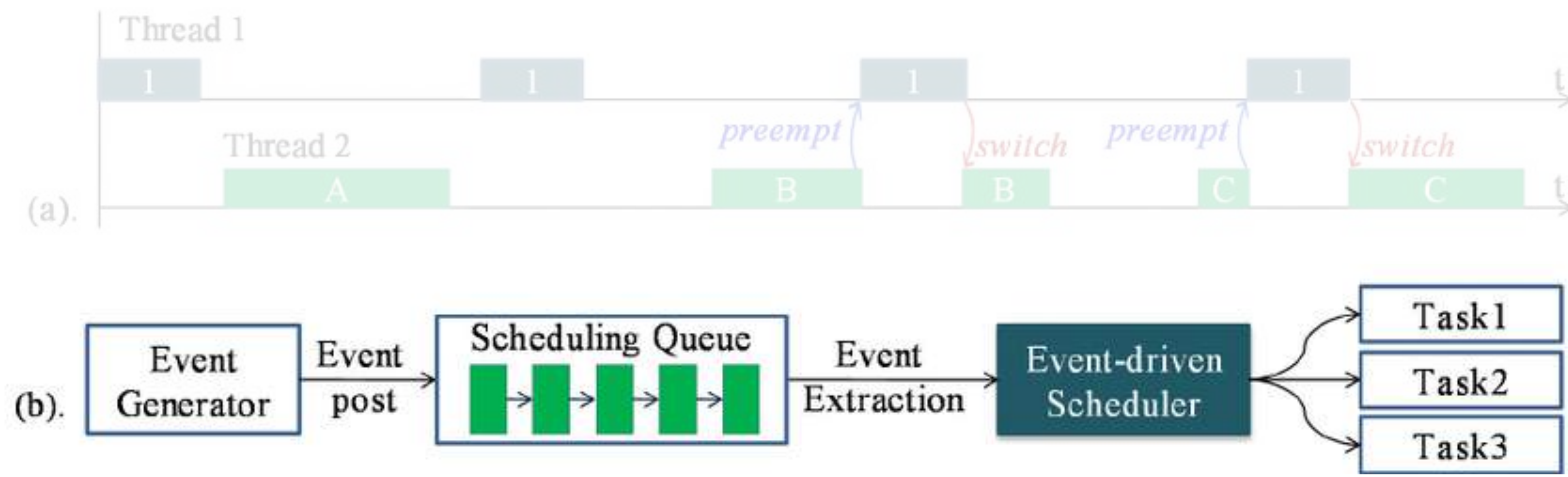
- All triggered events are queued and handled one by one; cannot preempt each other so only one run-time stack



- Software organized as event handlers
- Event handlers run to completion on each invocation

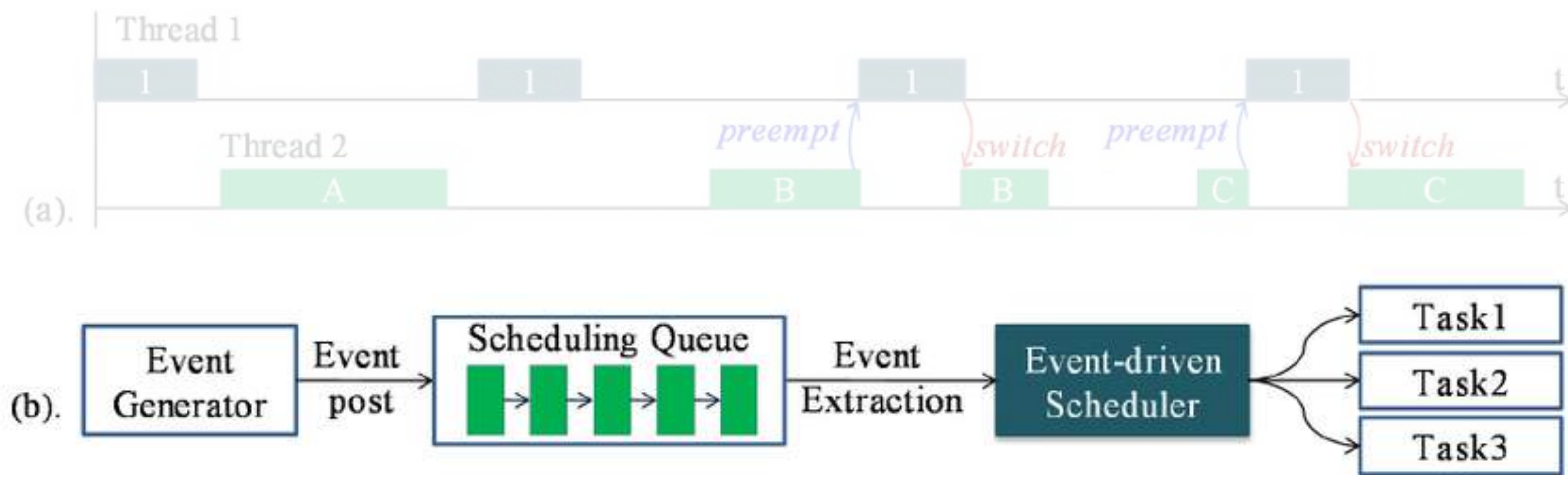


Which approach is more memory efficient?



Courtesy: ECE M119 UCLA Spring 2019

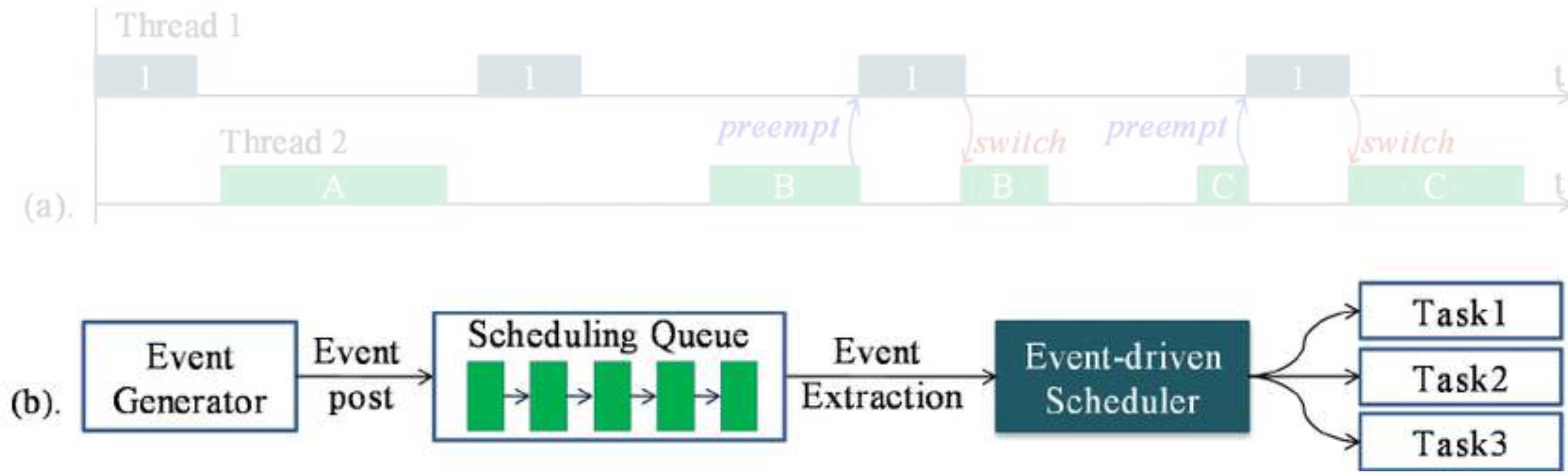
Which approach requires less context switching?



Courtesy: ECE M119 UCLA Spring 2019

Which approach does not require locking?

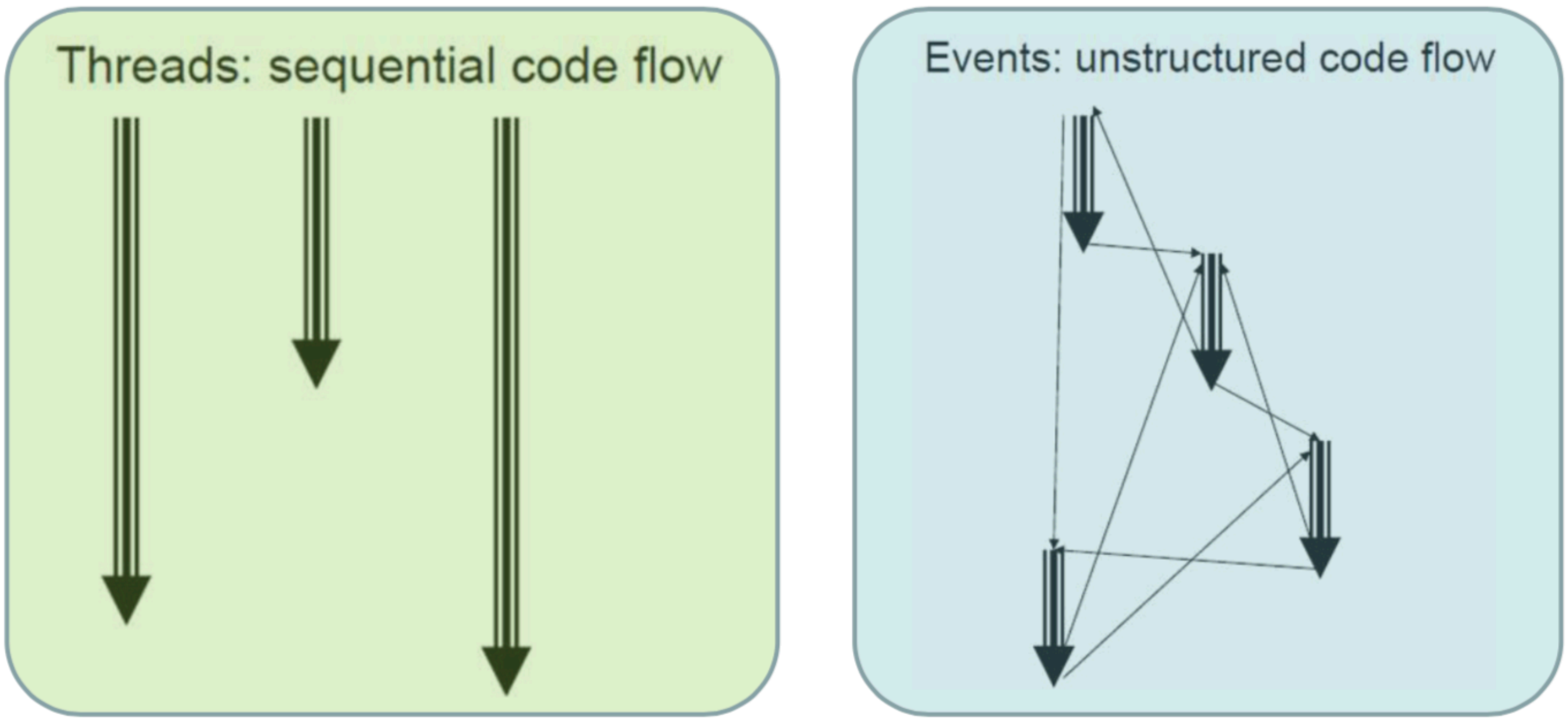
Needs locks, large context switch overhead, high memory consumption



No preemption possible, unstructured code flow and hard to follow, not suitable for long running tasks e.g. cryptography

Courtesy: ECE M119 UCLA Spring 2019

Multi-threaded v/s Event Driven



Boano & Calvi, "Introduction to Contiki", 2011

Multi-threaded v/s Event Driven: Example

- Consider an embedded system with a bunch of sensors
- For sensor #i
  - Receive samples
  - Process window of n_i events (may take time >> shortest sample interval)
  - Send result as a packet via a radio
- Thread approach: pair of threads per sensor
  - Reader thread that receives samples, assembles a window of samples, and passes that window to a Processor thread that processes the window samples, and send the result to radio via blocking I/O
- Event approach
  - Handle events from sensors and radio
  - Split processing into short enough chunks
  - Use radio in non-blocking I/O (request-done) and keep track of radio state

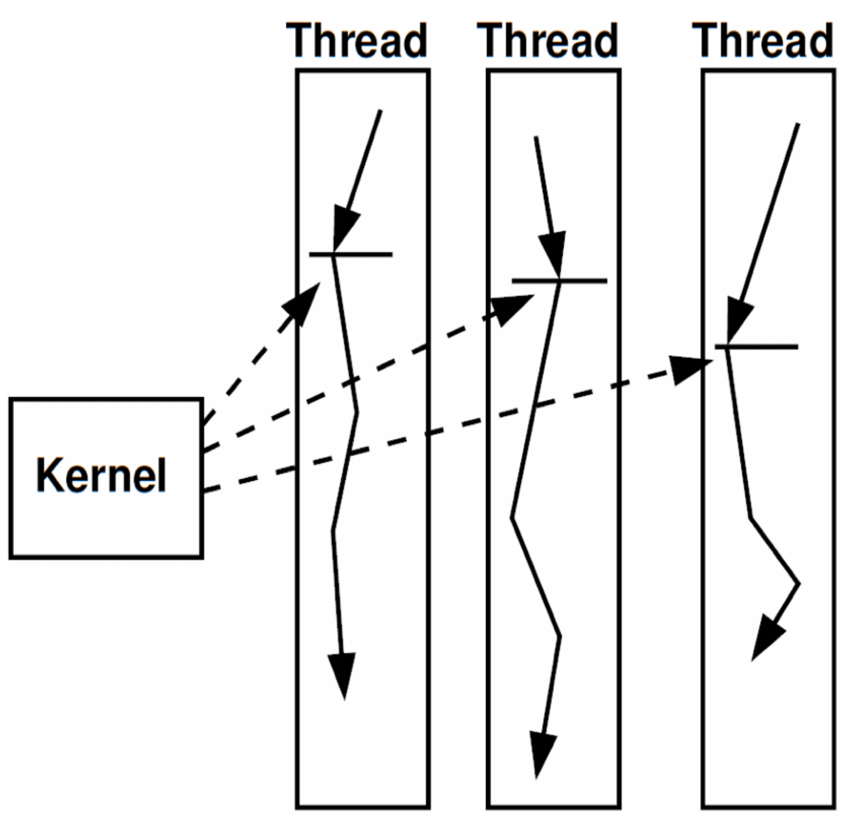
Comparison

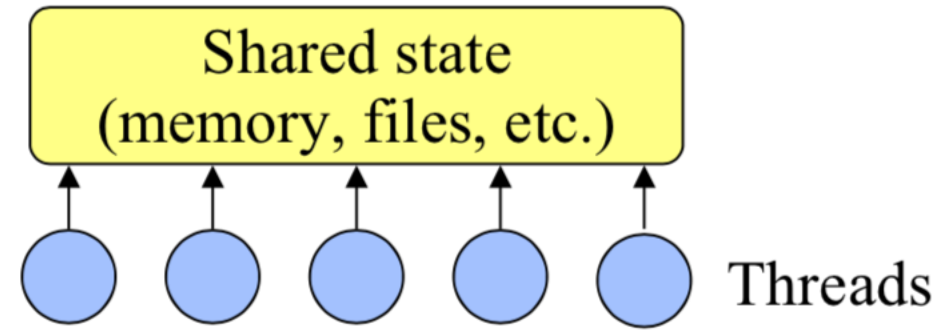
Multi-threaded	Event-driven Programming
• Easier to write and understand • State transition maintained in stack • Provide true concurrency • Scalable across multiple CPU cores	• Memory Efficient • Faster on single CPU (no context switching, locking) • More portable • Timing dependencies only related to events, not to internal scheduling
• Problems with synchronization, deadlock, and shared resources • Often not supported by the underlying software platform particularly in resource-constrained embedded systems • Hard to get good performance ‣ simple locking yields low concurrency • Overhead: multiple stacks, context switches	• Hard to write, understand, analyze, and debug • Long-running handlers affect responsiveness • Can't maintain local state across events (handler must "return") - Leads to "stack ripping": stacks must be manually reconstructed in the heap and passed as extra parameter from callback to callback • No CPU concurrency - handlers must run in sequence as they all share global state, challenge to take advantage of multiple cores

Threads Programming

Threads

- General-purpose solution for managing concurrency
  - OS: one kernel thread for each user process
  - Scientific applications: one thread per CPU
  - Distributed systems: process requests concurrently (overlap I/Os)
  - GUIs: threads correspond to user actions; can service display during long-running computations; multimedia, animations
- Multiple independent execution streams
  - Shared state
  - Synchronization (e.g. locks, conditions)
  - Preemptive scheduling





POSIX Threads

- P*OSIX threads* (*Pthreads*) in Linux is a set of C functions, types, and constants that help implement threading within your C applications
- Include the **`pthread.h`** header file and use the **`-pthread`** flag when compiling and linking the code using gcc in your makefiles

```
gcc -o test example01.c -l pthread
```

- All the Pthread functions are prefixed with `pthread_`
  - *`pthread_create`*: creates thread, OS may or may not start execution immediately
  - *`pthread_join`*: wait for thread to finish execution
  - *`pthread_exit`*: always finish main program with this function so program doesn't finish when main program finishes

POSIX Threads: C code example

```c
/* Threading application */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h> // Header file for thread library

// A normal C function that will be executed as a thread
void* threadFunction(void *var){
    // casting input argument type to integer
    int* input = (int *) var;
    // pause the program for 1 second
    sleep(1);
    printf("Received %d inside thread argument.\n", *input);
    return NULL;
}

int main() {
    int data = 6;
    // instantiating argument required for thread creation
    pthread_t thread_id;
    printf("Before Thread\n");
    // first argument requires reference of pthread_t variable, this variable can be used to manipulate the
created thread in future
    // second argument can be used to set thread related settings, we will use default setting and pass NULL
    // third argument is the function name which will be executed once the thread starts
    // fourth argument is the pointer variable that will be passed as input to the thread function named
"threadFunction"
    pthread_create(&thread_id, NULL, threadFunction, (void*)(&data) );
    // blocks the main function (thread) until the thread function is preempted
    pthread_join(thread_id, NULL);
    printf("After Thread\n");
    // release the thread once finisihed
    pthread_exit(0);
}
```

Input argument to thread functions is a void pointer

Cast the input pointer to desired data type pointer

Always initialize a thread ID

Create and join a thread with its ID and thread function name

Don’t forget to release thread

POSIX Threads: makefile and output example

This flag is very important to compile threads!!

```
CC=gcc
CFLAGS=-lpthread -I.

test: thread_sample.c
	$(CC) -o test thread_sample.c $(CFLAGS)
.PHONY: clean
clean:
	rm -f test
```

Output

```
debian@beaglebone:~/ece231/thread_code$ ./test
Before Thread
Received 6 inside thread argument.
After Thread
```

Reading

- Concurrency chapter at this link, https://pages.cs.wisc.edu/~remzi/OSTEP/
- Textbook: Read Chapter#6
  - POSIX Threads
  - Callback Functions