Long-Running Tasks and Basic Animation

01219116 Computer Programming II

Chaiporn Jaikaeo

Department of Computer Engineering
Kasetsart University

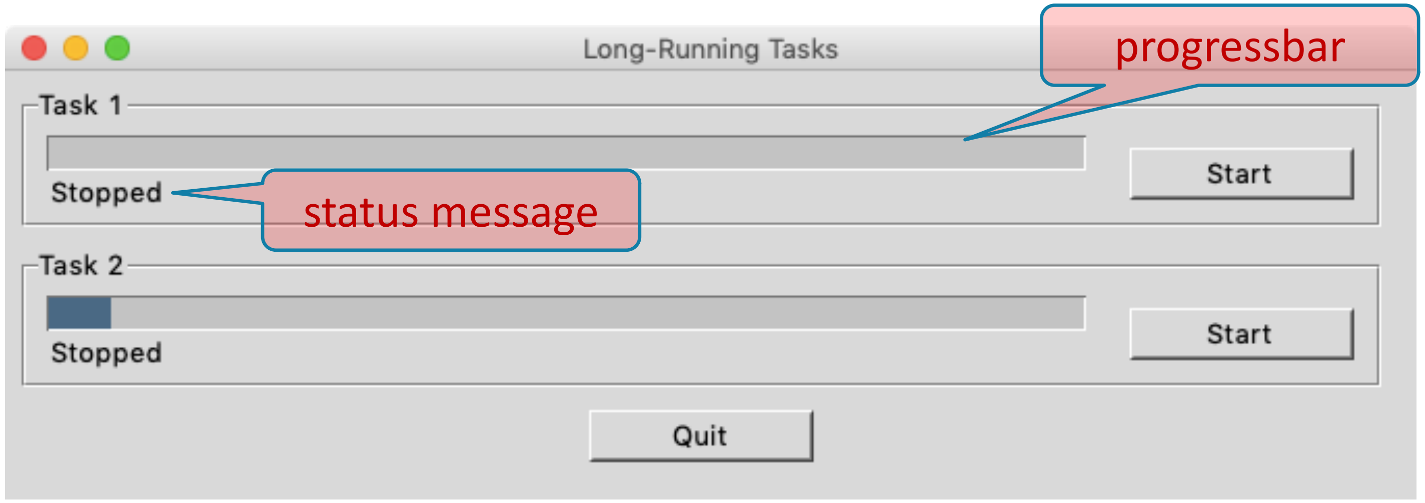
Revised 2021-02-15

Outline

- Handling long-running tasks
- Basic animation

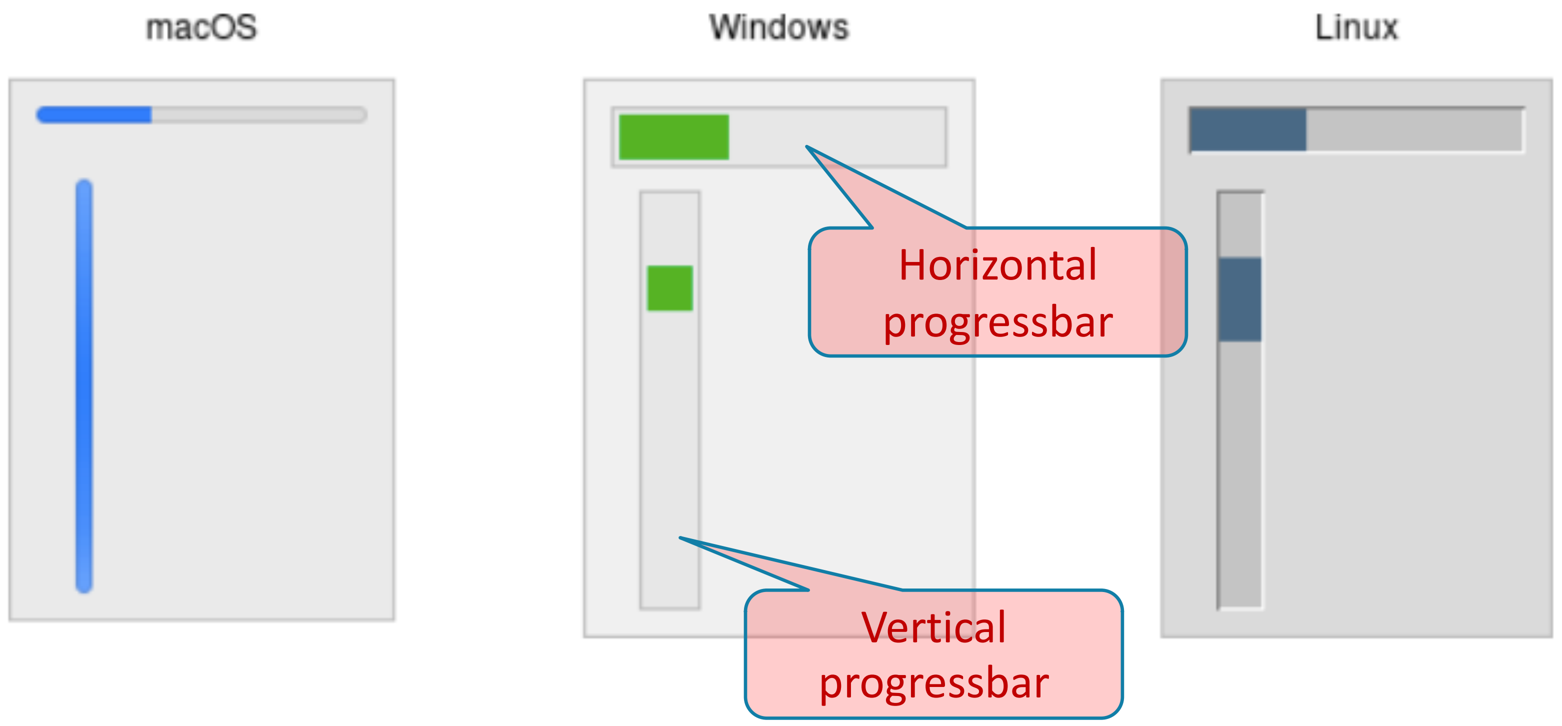
Example Scenario

- Consider two long-running tasks
  - Task 1: takes 10 seconds to complete
  - Task 2: completion time is unknown
- User clicks a button to start a specific task and is given a visual feedback that the task is running



Adding Feedback: Progressbar

- A Progressbar widget provides feedback to users about the progress of a lengthy operation



Progress Modes

- ***Determinate* mode**
  - The bar indicates relative progress towards completion
  - Set `mode` option to `"determinate"`
  - Use `maximum` option to specify the total number of steps (floating-point allowed), defaulted to 100.0
  - Set `variable` option to link with a control variable
- ***Indeterminate* mode**
  - The bar only shows the operation is in progress
  - Set `mode` option to `"indeterminate"`
  - Call `start()` method to start the progress animation; call `stop()` method to stop

Application Stub

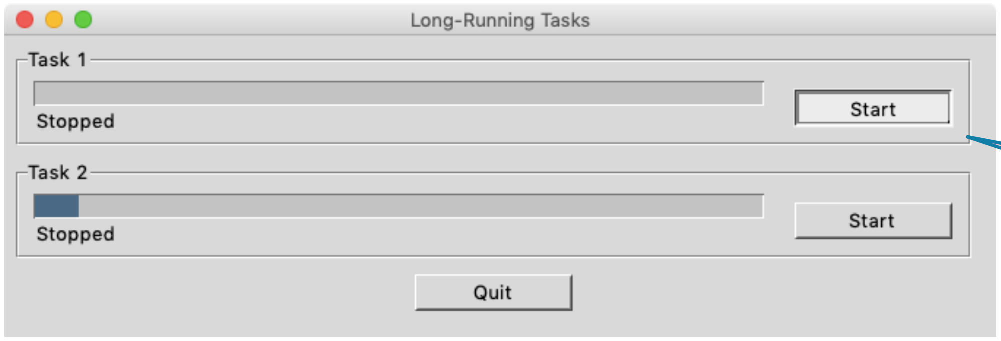
```
import time
import random
import tkinter as tk
from tkinter import ttk

class App(ttk.Frame):

    def __init__(self, parent):
        super().__init__(parent, padding="3 3 12 12")
        self.style = ttk.Style()
        self.style.theme_use("alt")
        parent.rowconfigure(0, weight=1)
        parent.columnconfigure(0, weight=1)
        self.grid(row=0, column=0, sticky="NEWS")
        self.create_widgets()

    def create_widgets(self):
        self.rowconfigure(0, weight=1)
        self.rowconfigure(1, weight=1)
        self.columnconfigure(0, weight=1)

        # control variables
        self.progress1 = tk.IntVar()
        self.status1 = tk.StringVar()
        self.status2 = tk.StringVar()

        # subframe for Task 1
        self.frame1 = ttk.LabelFrame(self, text="Task 1")
        self.frame1.grid(row=0, column=0, sticky="news", padx=5, pady=5)
        self.frame1.rowconfigure(0, weight=1)
        self.frame1.rowconfigure(1, weight=1)
        self.frame1.columnconfigure(0, weight=1)
        self.bar1 = ttk.Progressbar(self.frame1, length=500,
            variable=self.progress1, mode="determinate")
        self.label1 = ttk.Label(self.frame1, text="Stopped",
            textvariable=self.status1)
        self.start1 = ttk.Button(self.frame1, text="Start",
            command=self.run_task1)
        self.bar1.grid(row=0, column=0, sticky="sew", padx=10)
        self.label1.grid(row=1, column=0, sticky="wn", padx=10)
        self.start1.grid(row=0, column=1, rowspan=2, padx=10, pady=10)

        # subframe for Task 2
        self.frame2 = ttk.LabelFrame(self, text="Task 2")
        self.frame2.grid(row=1, column=0, sticky="news", padx=5, pady=5)
        self.frame2.rowconfigure(0, weight=1)
        self.frame2.rowconfigure(1, weight=1)
        self.frame2.columnconfigure(0, weight=1)
        self.bar2 = ttk.Progressbar(self.frame2, length=500,
            mode="indeterminate")
        self.label2 = ttk.Label(self.frame2, text="Stopped",
            textvariable=self.status2)
        self.start2 = ttk.Button(self.frame2, text="Start", c
            ommand=self.run_task2)
        self.bar2.grid(row=0, column=0, sticky="sew", padx=10)
        self.label2.grid(row=1, column=0, sticky="wn", padx=10)
        self.start2.grid(row=0, column=1, rowspan=2, padx=10, pady=10)

        self.quit = ttk.Button(self, text="Quit", command=root.destroy)
        self.quit.grid(row=2, column=0, padx=5, pady=5, sticky="s")

        self.status1.set("Stopped")
        self.status2.set("Stopped")

    def run_task1(self):
        print("Running task 1...")
        self.task1()

    def run_task2(self):
        print("Running task 2...")
        self.task2()

    def task1(self):   # simulate determinate long-running task
        time.sleep(10)

    def task2(self):   # simulate indeterminate long-running task
        time.sleep(random.randrange(5,10))

if __name__ == "__main__":
    root = tk.Tk()
    root.title("Long-Running Tasks")
    app = App(root)
    root.mainloop()
```

Updating GUI: First Attempt

- Just consider **Task 1** for now
- This code is still not working properly

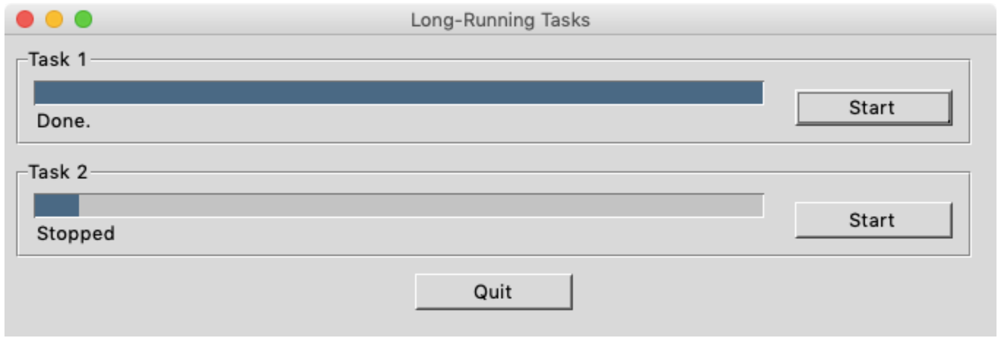
```
    def task1(self):   # simulate determinate long-running task
        self.start1.config(state="disabled")  # disable Start button
        self.status1.set("Running...")
        for i in range(101):
            time.sleep(0.1)
            self.progress1.set(i)
        self.start1.config(state="enabled")  # re-enable button
        self.status1.set("Done.")
```

Problem: Unresponsive GUI



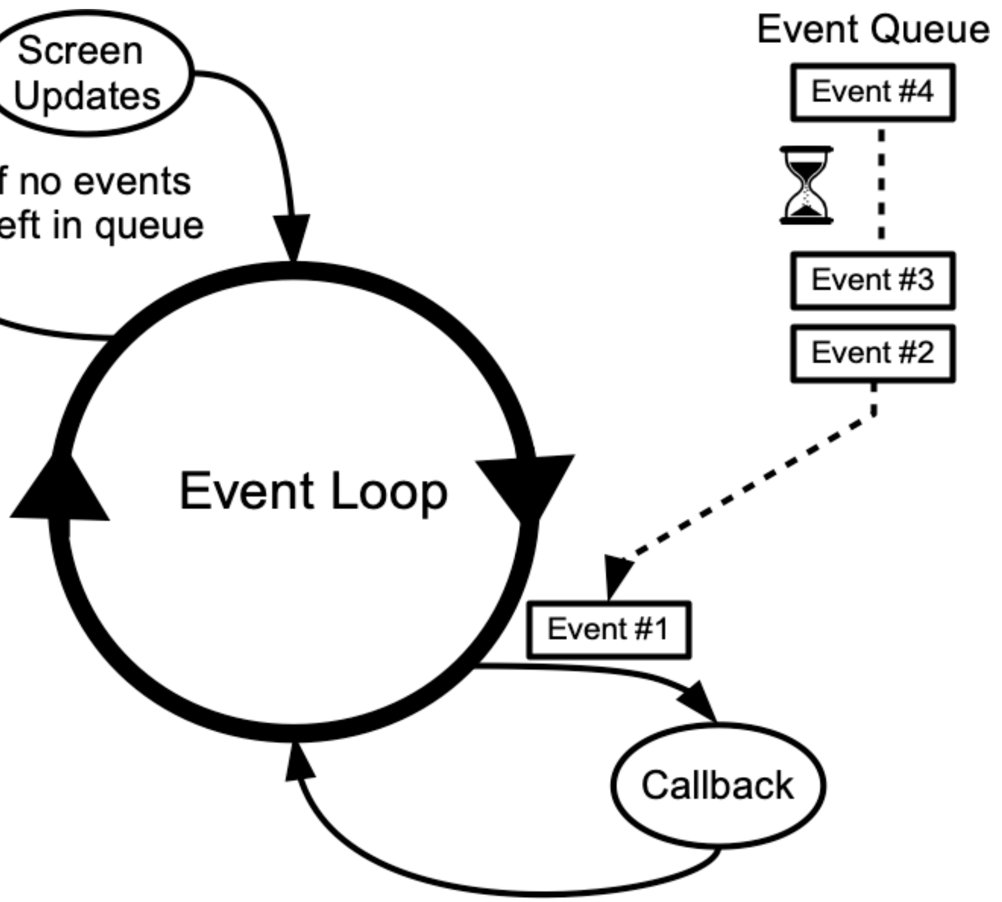
GUI not getting updated at all

After clicking "Start" for Task 1



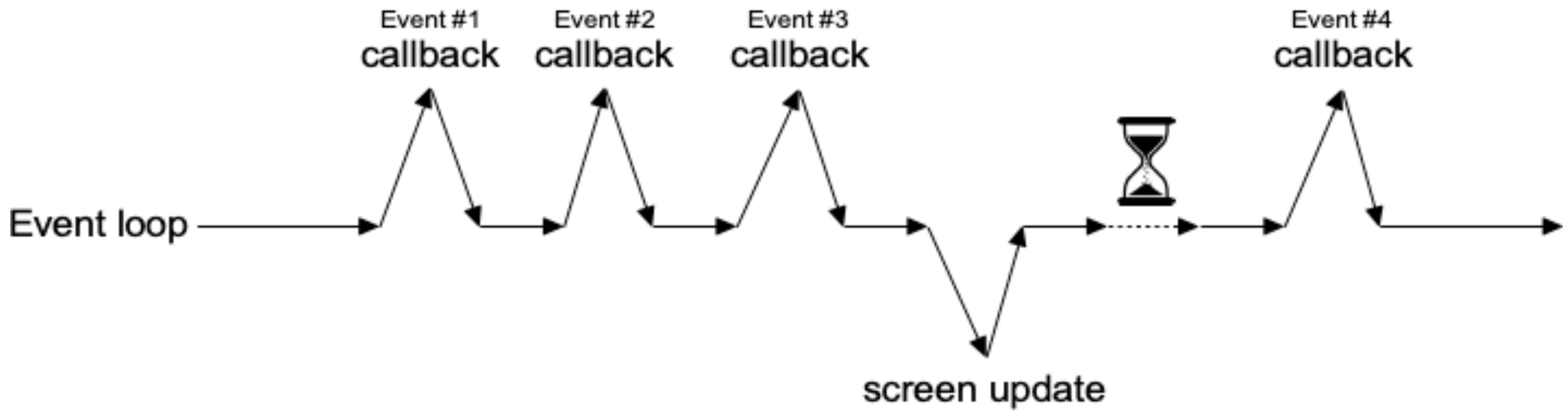
Ten seconds later

Event Loop Revisited



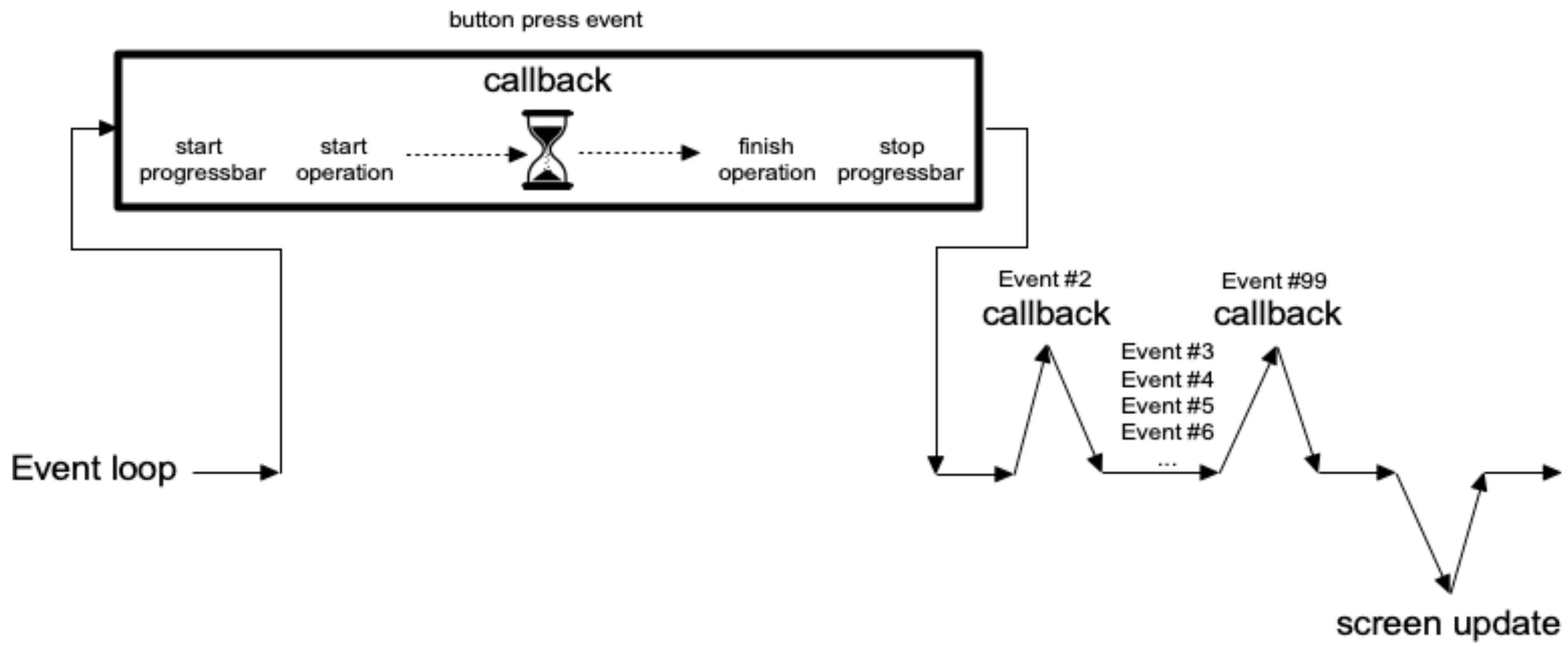
Event Loop: Desired Behaviour

- Callbacks are kept as short as possible
- GUI screen is updated regularly, giving a responsive behaviour



Event Loop: Blocking

- A callback takes too long to complete
  - It *blocks* the entire event loop, causing a very long delay before the next screen update



Calling **update()** Manually

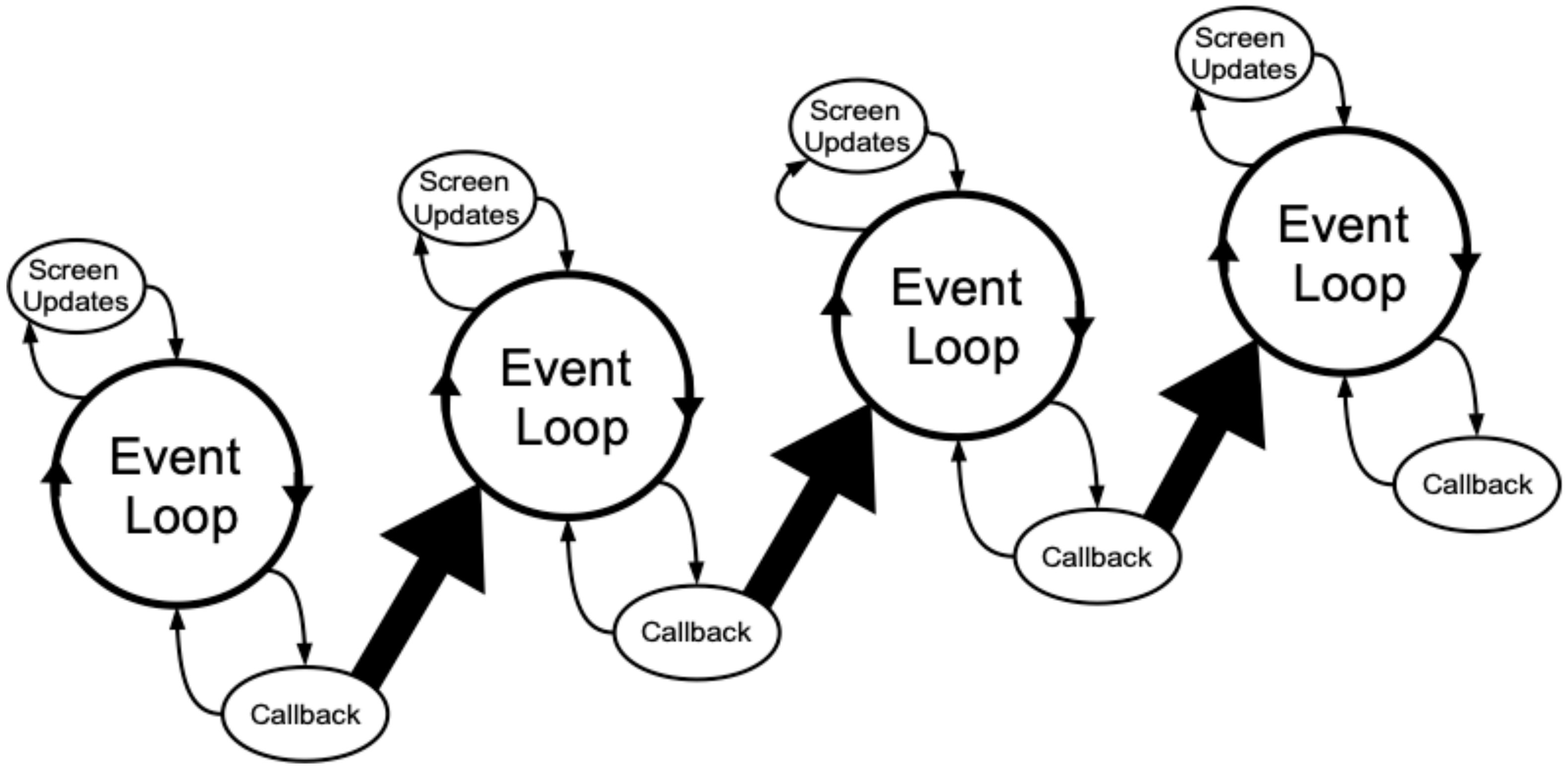
- The **update()** method allows the GUI screen to update

```
    def task1(self):   # simulate determinate long-running task
        self.start1.config(state="disabled")  # disable Start button
        self.status1.set("Running...")
        self.update()
        for i in range(101):
            time.sleep(0.1)
            self.progress1.set(i)
            self.update()
        self.start1.config(state="enabled")  # re-enable button
        self.status1.set("Done.")
```

- Problem: this could end up with nested event processing

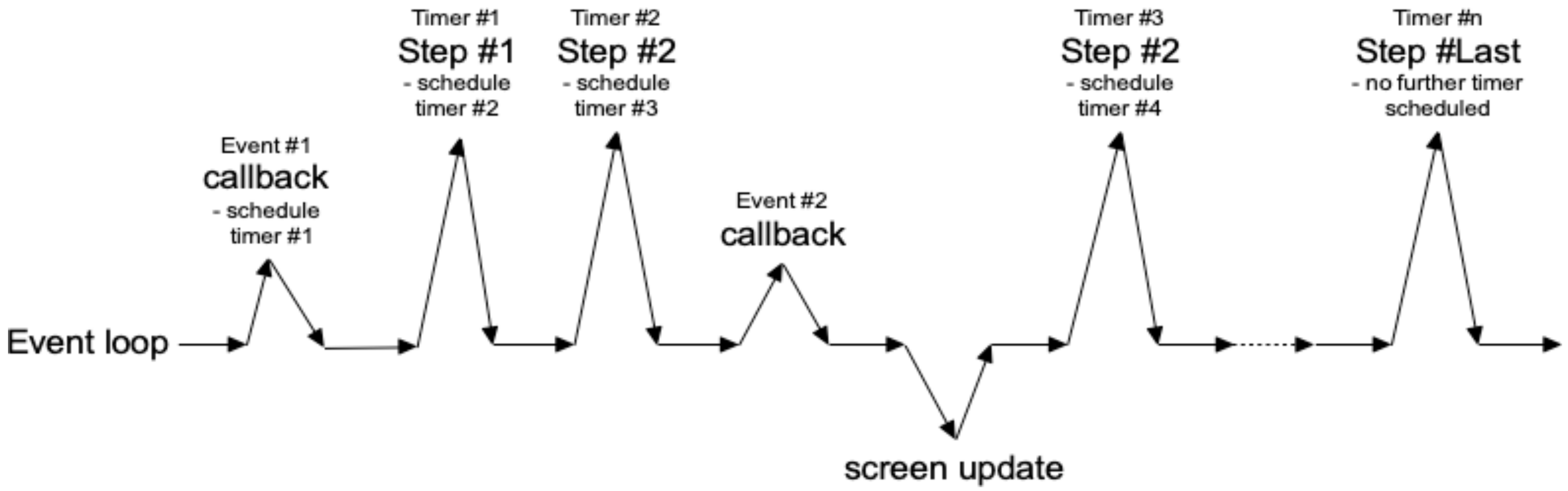
Nested Event Processing

- Each call to **`update()`** effectively starts a new event loop nested within the existing one
- New event loops could be created indefinitely



Split Operation

- Use the `after()` method to split the event handler into multiple ones



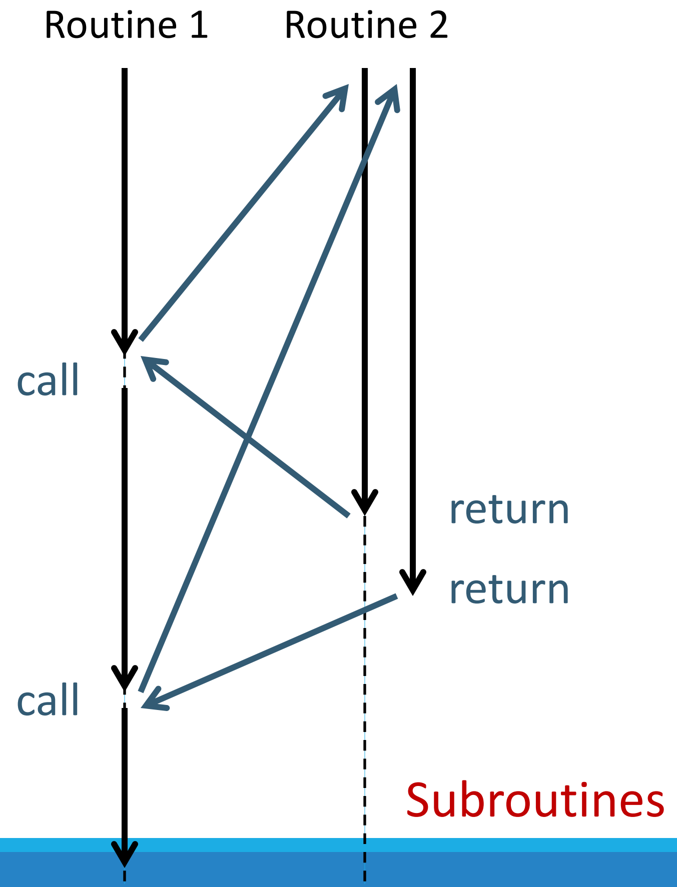
Split Operation: Code

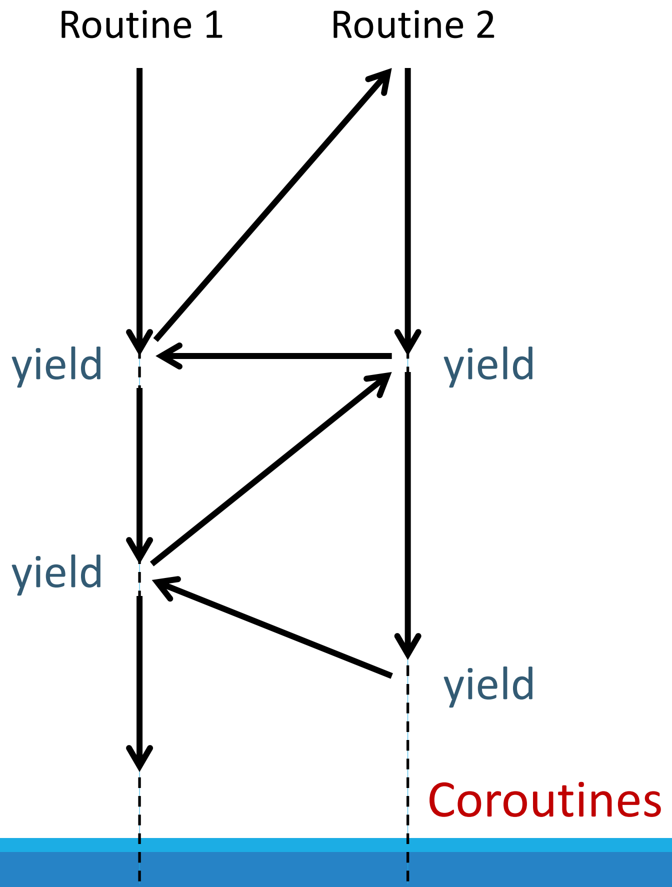
```python
    def task1(self):
        self.start1.config(state="disabled")  # disable Start button
        self.status1.set("Running...")
        self.after(10, lambda: self.task1_step(0))

    def task1_step(self, step):   # simulate determinate long-running task
        time.sleep(0.1)
        self.progress1.set(step)
        if step < 100:
            self.after(10, lambda: self.task1_step(step+1))
        else:
            self.task1_done()

    def task1_done(self):
        self.status1.set("Done.")
        self.start1.config(state="enabled")  # reenable Start button
```

- Observe that Task 1's code has been modified quite significantly from the original

Coroutines

- It is inconvenient to maintain states across "broken up" subtasks
- A coroutine can maintain continuations across each call





Python's Generator as Coroutine

- Python's generator can be used to implement a simple coroutine
- A generator is created from a function with one or more **yield** statements
  - A **yield** statement suspends the routine and returns back to the caller
  - A call to **next()** causes the routine to continue

```python
def routine1():
    print("Hello")
    yield
    print("How are you?")
    yield
    print("Goodbye")
```

```
>>> coro = routine1()
>>> coro
<generator object routine1 at 0x104d545f0>
>>> next(coro)
Hello
>>> next(coro)
How are you?
>>> next(coro)
Goodbye
```

Coroutine: Code

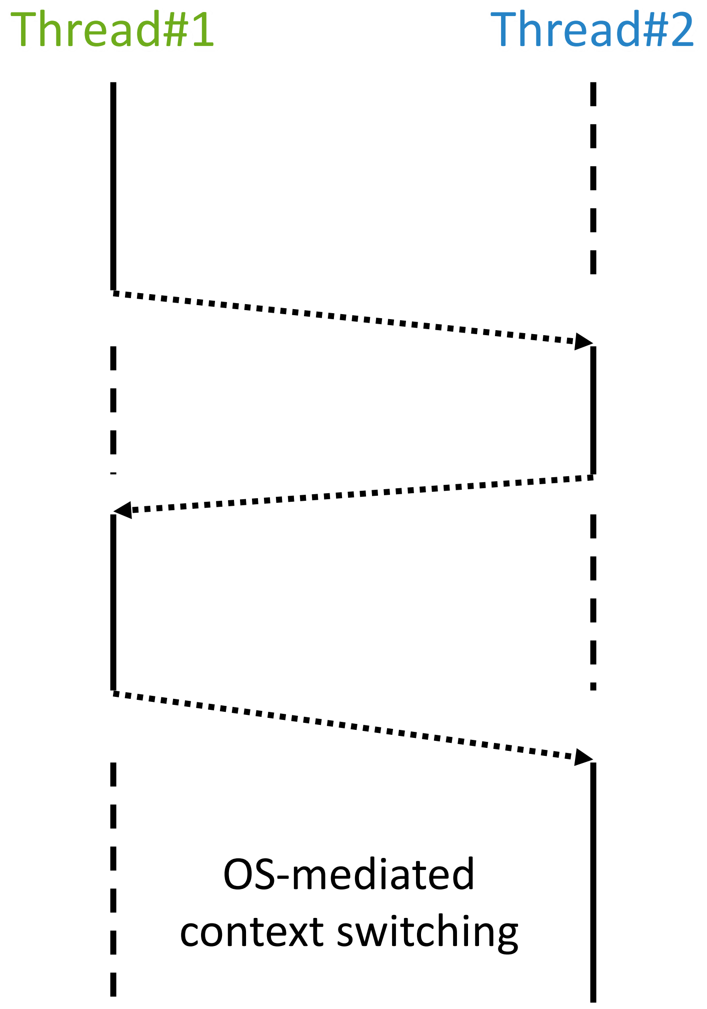
```python
def perform_task(self, coro, wait=10):  # generic "task executor"
    try:
        next(coro)
        self.after(wait, lambda: self.perform_task(coro, wait))
    except StopIteration:
        pass

def run_task1(self):
    print("Running task 1...")
    task1_coro = self.task1()
    self.perform_task(task1_coro)

def task1(self):   # simulate determinate long-running task
    self.start1.config(state="disabled")  # disable Start button
    self.status1.set("Running...")
    yield
    for i in range(101):
        time.sleep(0.1)
        self.progress1.set(i)
        yield
    self.start1.config(state="enabled")  # reenable Start button
    self.status1.set("Done.")
```

`StopIteration` exception is thrown when the coroutine completed its execution

Try comparing this code with the original one on page 7

Spawning a New Thread

- Some task cannot be broken up
  - E.g., calling a function written by other programmer
- A new ***thread*** of execution can be spawned to handle the long-running task
  - The event loop in the main thread will keep running
- WARNING: only the main thread is allowed to call GUI-related functions
  - Your application will crash when trying to update GUI from a different thread



Spawning a Thread: Code

- Suppose Task 2 cannot be broken up, so we must spawn a thread to run it

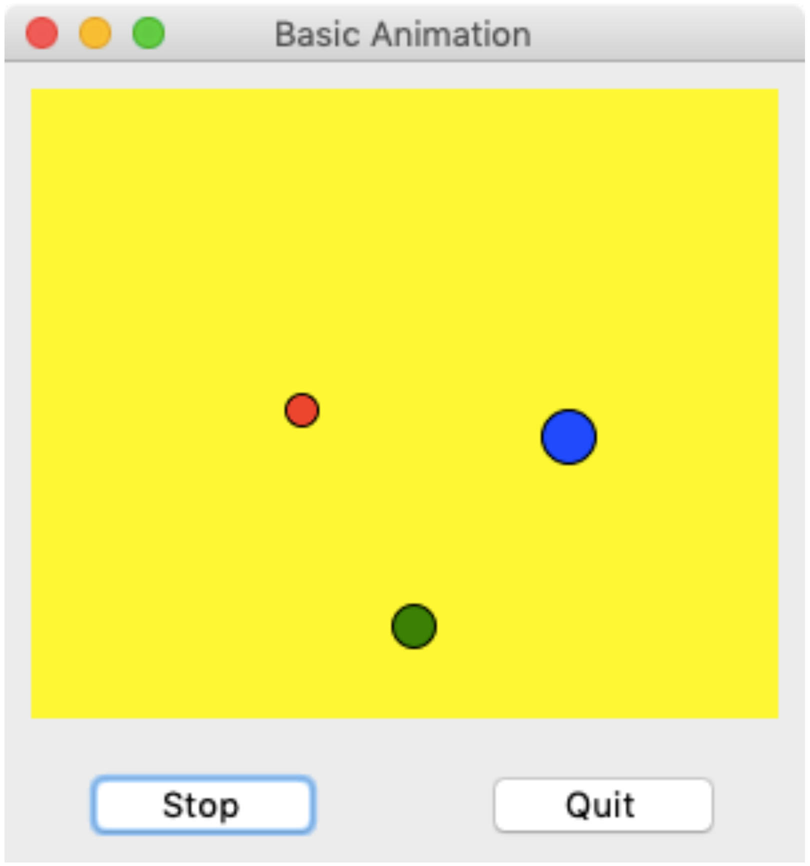
```
from threading import Thread
    :
    def run_task2(self):
        print("Running task 2...")
        self.task2_thread = Thread(target=self.task2)
        self.task2_thread.start()
        self.status2.set("Running...")
        self.start2.config(state="disabled")
        self.bar2.start()
        self.after(10, self.check_task2)

    def check_task2(self):   # check task 2 regularly whether it has completed
        if self.task2_thread.is_alive():
            self.after(10, self.check_task2)
        else:
            self.status2.set("Done.")
            self.start2.config(state="enabled")
            self.bar2.stop()

    def task2(self):   # simulate indeterminate long-running task
        time.sleep(random.randrange(5,10))
```

Basic Animation

Animation Techniques

- Loops with `time.sleep()` and manual `update()` calls
  - Not recommended; could end up with nested event loops
- Split operation with a timer, i.e., `after()` method
- Generators/coroutines
- Threads

Scenario: Balls Moving in Squares



Application Stub

```python
import time
import tkinter as tk
import tkinter.ttk as ttk

class App(ttk.Frame):

    def __init__(self, parent):
        super().__init__(parent)
        parent.rowconfigure(0, weight=1)
        parent.columnconfigure(0, weight=1)
        self.grid(row=0, column=0, sticky="news")
        self.rowconfigure(0, weight=1)
        self.columnconfigure(0, weight=1)
        self.columnconfigure(1, weight=1)
        self.create_widgets()
        self.is_animating = False

    def create_widgets(self):
        self.canvas = tk.Canvas(self, borderwidth=0,
            highlightthickness=0, bg="yellow")
        self.canvas.grid(row=0, column=0, columnspan=2,
            sticky="news", padx=10, pady=10)
        self.btn_animate = ttk.Button(self, text="Animate",
            command=self.toggle_animation)
        self.btn_animate.grid(row=1, column=0, pady=10)
        ttk.Button(self, text="Quit", command=root.destroy).grid(
            row=1, column=1, pady=10)
```

```python
    def toggle_animation(self):
        self.is_animating = not self.is_animating
        if self.is_animating:
            self.btn_animate.config(text="Stop")
            self.animate()
        else:
            self.btn_animate.config(text="Animate")

    def animate(self):
        # put animation update code here
        #   :

        # schedule the next update
        if self.is_animating:
            self.after(33, self.animate)


if __name__ == "__main__":
    root = tk.Tk()
    root.title("Basic Animation")
    root.geometry("300x300")
    app = App(root)
    root.mainloop()
```

Call itself every 33 ms
(~30 updates/sec)

The Ball Class

- Let us define the **Ball** class that keeps track of a ball's states, e.g., size, color, current position, movement
- The **update()** method is expected to be called periodically to update the ball's states

```
class Ball:

    def __init__(self, canvas, size, color, speed):
        self.canvas = canvas
        self.size = size
        self.id = canvas.create_oval(0, 0, size, size, fill=color)
        self.speed = speed

    def move(self, x, y):
        self.canvas.moveto(self.id, x-self.size/2, y-self.size/2)

    def update(self):
        # ball update code goes here
```

Adding Ball's Animation

- We will implement the ball's animation process as a coroutine

```
class Ball:

    def __init__(self, canvas, size, color, speed):
        :
        self._coro = self.animate()

    def update(self):
        next(self._coro)

    def animate(self):  # move the ball in a square
        while True:
            for x in range(100, 200, self.speed):
                self.move(x, 100)
                yield
            for y in range(100, 200, self.speed):
                self.move(200, y)
                yield
            for x in range(200, 100, -self.speed):
                self.move(x, 200)
                yield
            for y in range(200, 100, -self.speed):
                self.move(100, y)
                yield
```

Creating and Updating Balls

```
class App(ttk.Frame):

    def __init__(self, parent):
        :
        self.balls = [
            Ball(self.canvas, 12, "red",   3),
            Ball(self.canvas, 16, "green", 2),
            Ball(self.canvas, 20, "blue",  1),
        ]

    def animate(self):
        for ball in self.balls:
            ball.update()

        # schedule the next update
        if self.is_animating:
            self.after(33, self.animate)
```

Conclusion

- A long-running task inside an event handler must be avoided; otherwise the GUI will freeze
- Available options
  - Calling widget's `update()` method manually
  - Breaking the long-running task into several shorter tasks and wire them up with timers, i.e., `after()` method
  - Converting the long-running task into a coroutine
  - Running the long-running task in a separate thread
- Animation is considered a long-running task that updates the screen periodically

References

- TkDocs Tutorial: Event Loop
  - https://tkdocs.com/tutorial/eventloop.html
- Python Programming for the Absolute Beginner, 3rd Edition: Time Tools, Threads, and Animation
  - https://flylib.com/books/en/2.723.1/time_tools_threads_and_animation.html