1, October 2024

CSE, IIT KGP

Systems Programming: Threading (Part 1)

Computing Lab (CS 69201)

Department of Computer Science and Engineering

IIT Kharagpur

What is systems program?

- Programming that requires close interaction with the operating system

What is systems program?

- Programming that requires close interaction with the operating system

- Our target areas:
  - Operating Systems
  - Networking Software
  - Compilers & Interpreters … so on.

What is systems program?

- Programming that requires close interaction with the operating system

- Our target areas:
  - Operating Systems
  - Networking Software
  - Compilers & Interpreters … so on.

- Why are we even studying it ?
  - Forms the foundation of all software
  - Critical for performance-sensitive applications
  - Enables efficient use of hardware resources

Quick Recap

- **Processes**
  - Independent units of execution
  - Have their own memory space
  - Created through system calls (e.g., fork() in Unix)
  - Communicate via Inter-Process Communication (IPC)

Quick Recap

- **Processes**
  - Independent units of execution
  - Have their own memory space
  - Created through system calls (e.g., fork() in Unix)
  - Communicate via Inter-Process Communication (IPC)

- **Inter-Process Communication (IPC)**
  - Methods for processes to exchange data
  - Includes pipes, shared memory,

Introduction to Concurrency

- **What is Concurrency?**
  - Ability to handle multiple tasks simultaneously. Sure ?

Introduction to Concurrency

- **What is Concurrency?**
  - Ability to handle multiple tasks simultaneously. Sure ?

- **Need of Concurrency in System Programming?**
  - Utilizes multi-core processors effectively
  - Improves system responsiveness
  - Enables parallel processing of tasks

Introduction to Concurrency

- **What is Concurrency?**
  - Ability to handle multiple tasks simultaneously. Sure ?

- **Need of Concurrency in System Programming?**
  - Utilizes multi-core processors effectively
  - Improves system responsiveness
  - Enables parallel processing of tasks

- **Challenges of Concurrency**
  - Race conditions
  - Deadlocks
  - Increased complexity in program design

Is concurrency same as Parallelism?

Is concurrency same as Parallelism?

- Parallelism is when tasks literally run at the same time, e.g., on a multicore processor.

- Parallelising CPU bound tasks improves throughput.

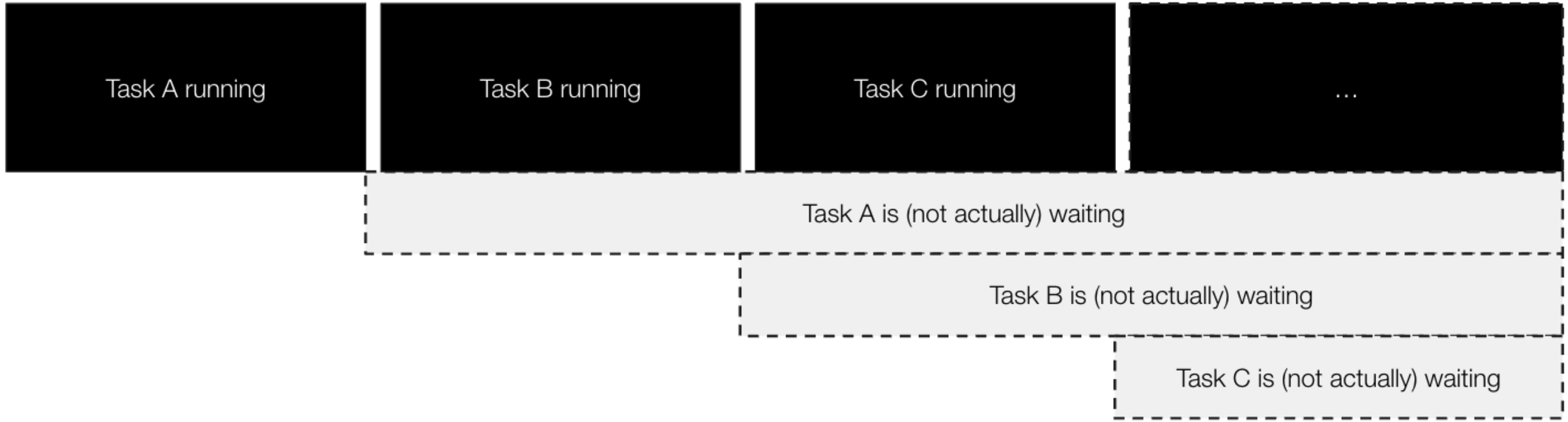
Is concurrency same as Parallelism?

- Parallelism is when tasks literally run at the same time, e.g., on a multicore processor.
- Parallelising CPU bound tasks improves throughput.
- But what if, tasks are blocked/idle most of the time?

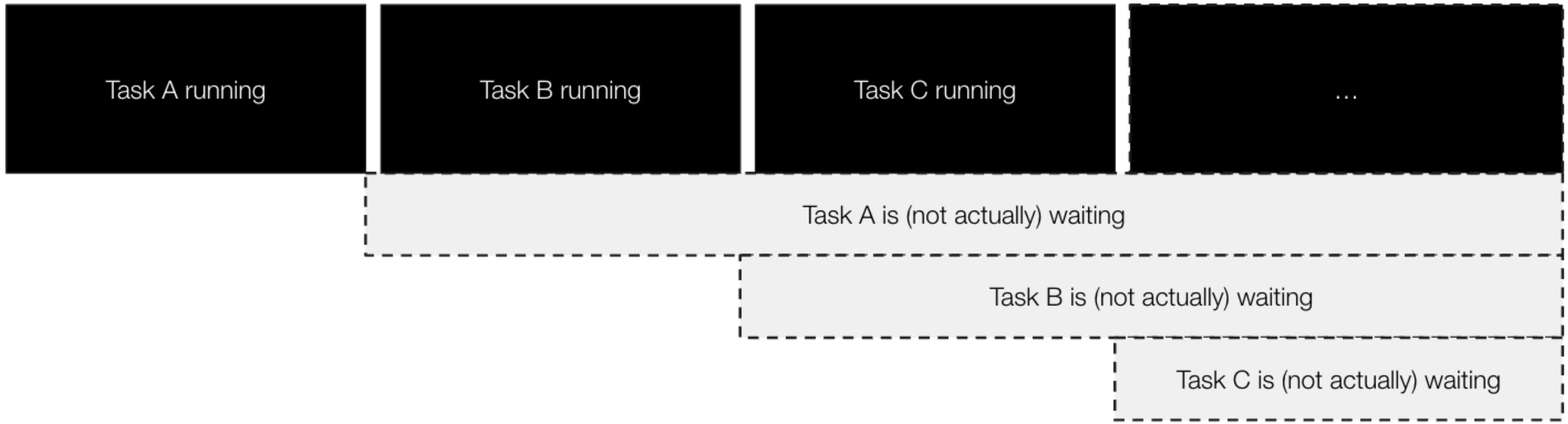
So parallelise it ?

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

The idea of concurrency

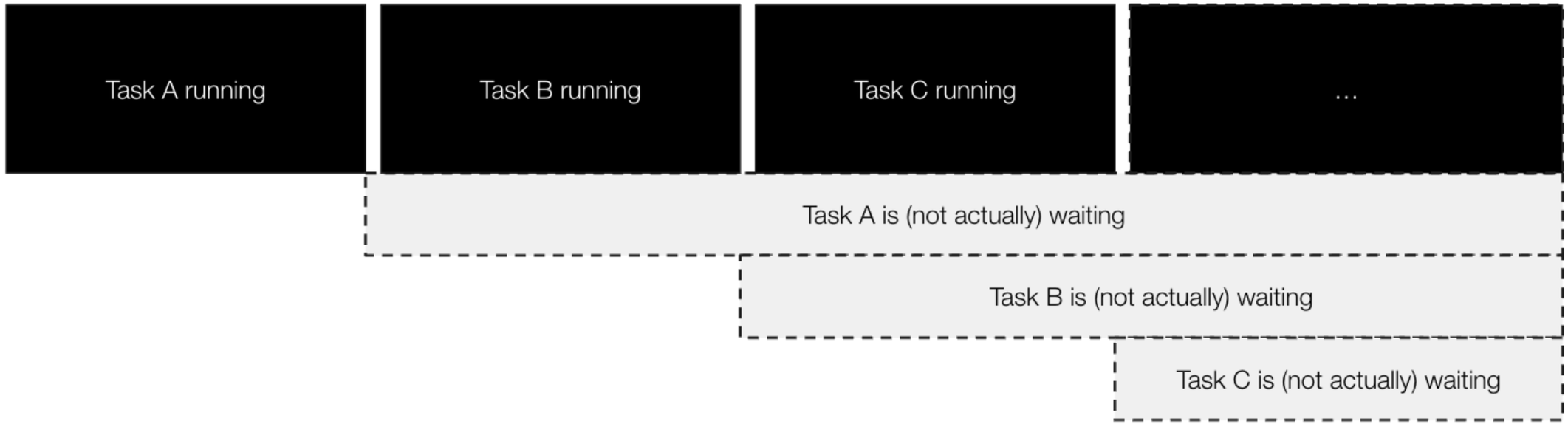


The idea of concurrency



- Concurrency is when two or more tasks can **start, run, and complete together** [doesn't necessarily mean they'll ever both be running at the same instant]
- For example, **multitasking on a single-core machine.**
- Concurrent IO bound and CPU bound tasks improve throughput

The idea of concurrency



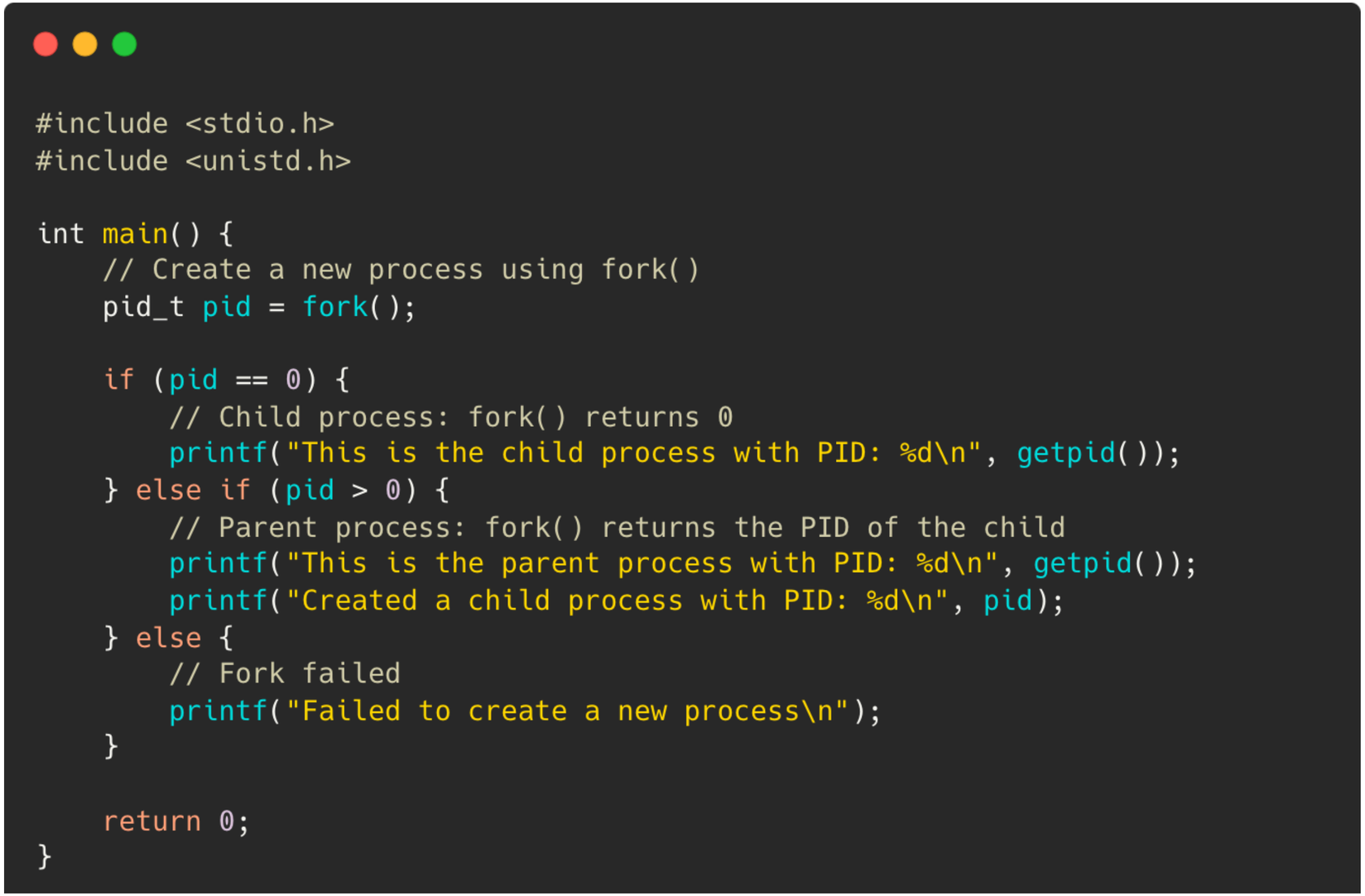
- Concurrency is when two or more tasks can **start, run, and complete together** [doesn't necessarily mean they'll ever both be running at the same instant]
- For example, **multitasking on a single-core machine.**
- Concurrent IO bound and CPU bound tasks improve throughput

do you find any catch here ?


This is where the fun begins.

How to create a process ?

How to create a process ?

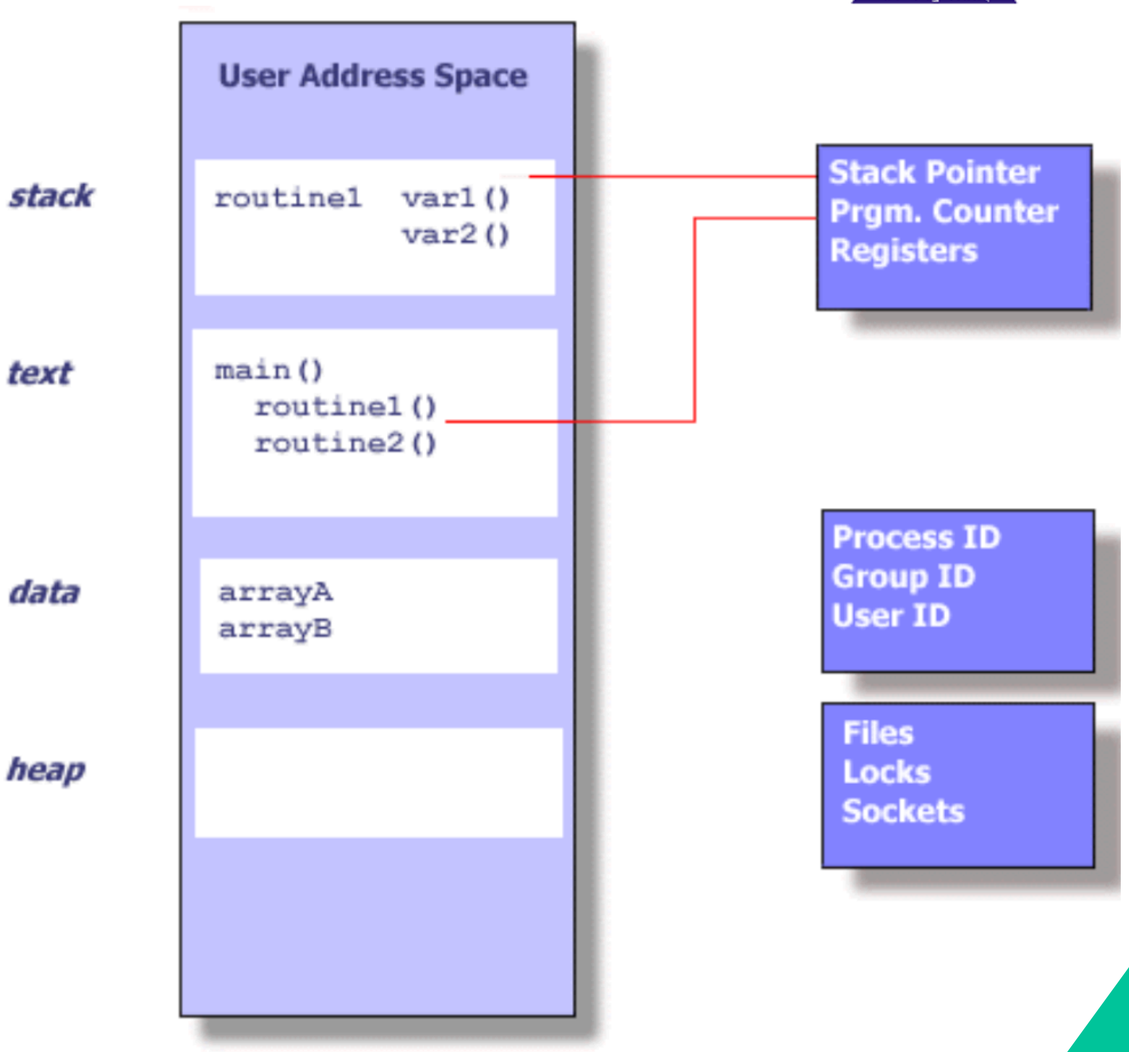


```c
#include <stdio.h>
#include <unistd.h>

int main() {
    // Create a new process using fork()
    pid_t pid = fork();

    if (pid == 0) {
        // Child process: fork() returns 0
        printf("This is the child process with PID: %d\n", getpid());
    } else if (pid > 0) {
        // Parent process: fork() returns the PID of the child
        printf("This is the parent process with PID: %d\n", getpid());
        printf("Created a child process with PID: %d\n", pid);
    } else {
        // Fork failed
        printf("Failed to create a new process\n");
    }

    return 0;
}
```

Process & Threads (1/4)

Processes contain information about program resources and program execution state, including:

- Process ID, process group ID, user ID, and group ID, address space
- Program instructions, registers, stack, heap.
- File descriptors, inter-process communication tools
- Shared libraries

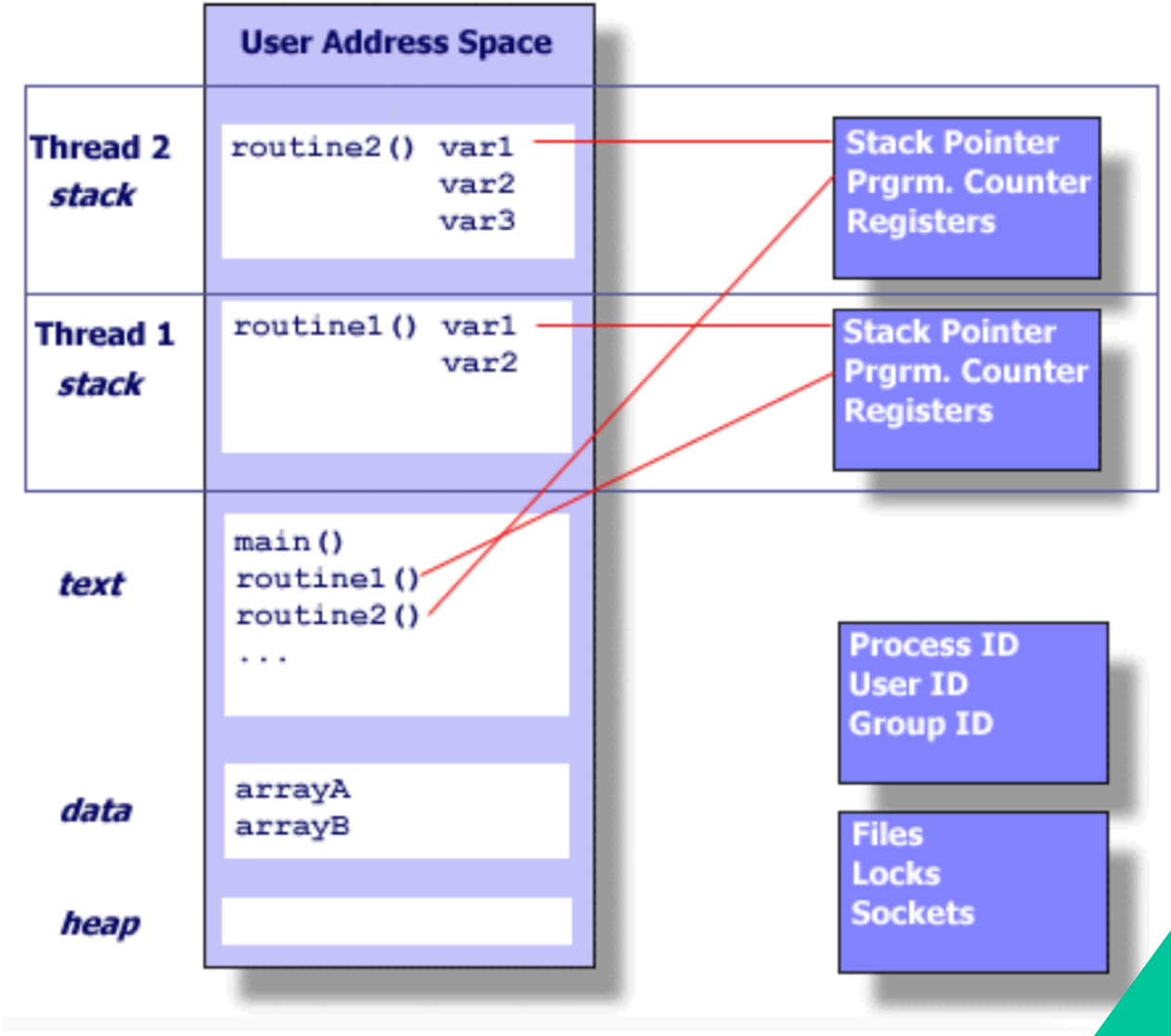


Process & Threads


```c
#include <stdio.h>
#include <unistd.h>

int main() {
    printf("Process ID: %d\n", getpid());           // Get Process ID
    printf("Parent Process ID: %d\n", getppid());   // Get Parent Process ID
    printf("User ID: %d\n", getuid());              // Get User ID
    printf("Group ID: %d\n", getgid());             // Get Group ID
    return 0;
}
```

Process & Threads (2/4)

Threads use and exist within these process resources, yet are able to be scheduled by the operating system and run as independent entities within a process

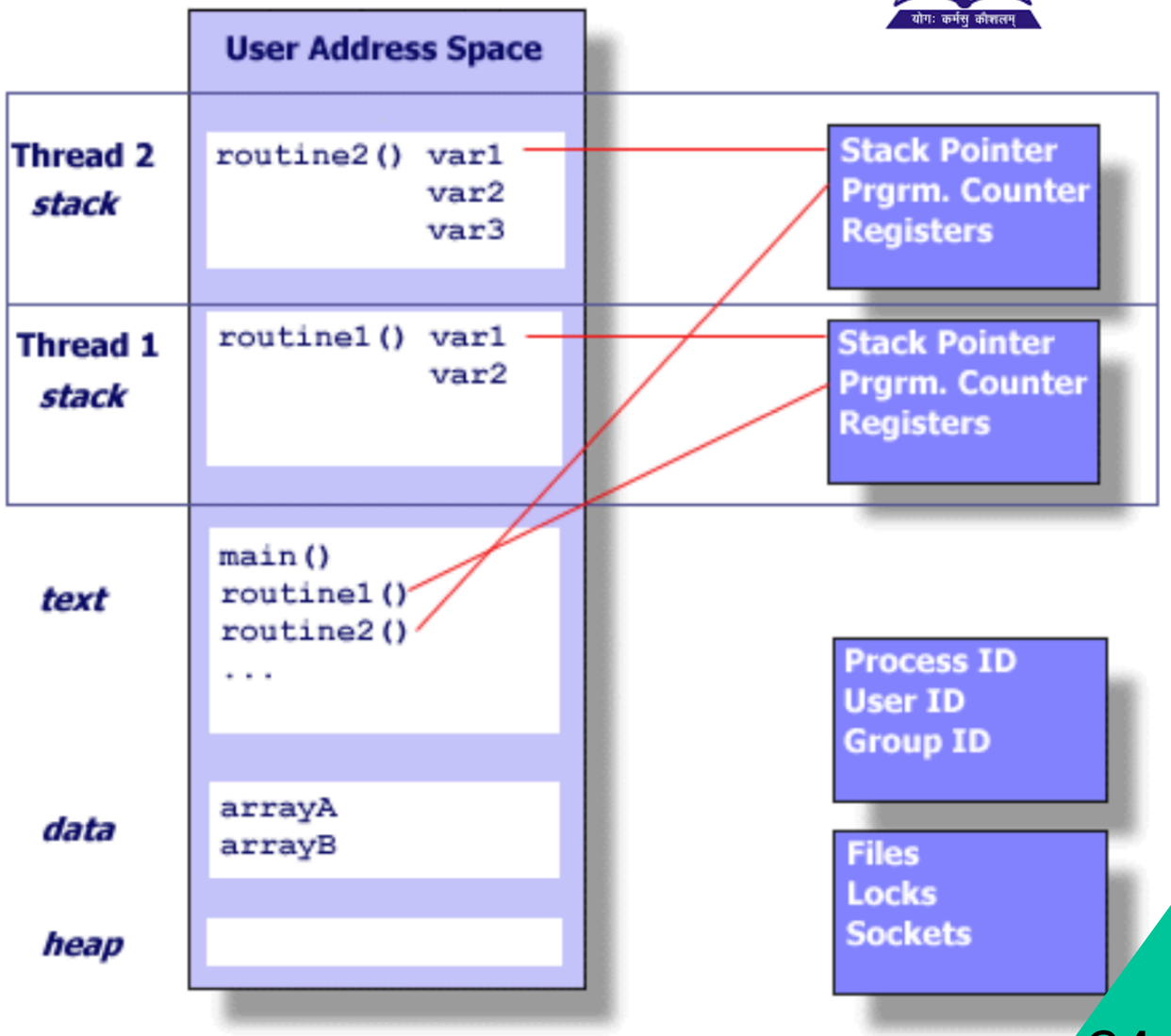


Process & Threads


```c
#include <stdio.h>
#include <pthread.h>

void* thread_function(void* arg) {
    printf("Thread running with ID: %ld\n", pthread_self());
    return NULL;
}

int main() {
    pthread_t thread;
    pthread_create(&thread, NULL, thread_function, NULL);  // Create a new
thread
    pthread_join(thread, NULL);  // Wait for thread to finish
    printf("Main process finished\n");
    return 0;
}
```

Process & Threads (3/4)

- A thread can possess an independent flow of control and be schedulable because it maintains its own:
  - Stack pointer
  - Registers
  - Scheduling properties
  - Set of pending and blocked signals
  - Thread specific data.



Process & Threads (4/4)

Because threads within the same process share resources:

- Changes made by one thread to shared system resources (such as closing a file) will be seen by all other threads .
- Two pointers having the same value point to the same data.
- Reading and writing to the same memory locations is possible, and therefore requires explicit synchronization by the programmer

Process & Threads


```c
#include <stdio.h>
#include <pthread.h>

int shared_resource = 0;

void* thread_function(void* arg) {
    shared_resource++;
    printf("Thread updated shared resource to %d\n", shared_resource);
    return NULL;
}

int main() {
    pthread_t thread1, thread2;
    pthread_create(&thread1, NULL, thread_function, NULL);
    pthread_create(&thread2, NULL, thread_function, NULL);

    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);

    printf("Final value of shared resource: %d\n", shared_resource);
    return 0;
}
```

pthreads

- For UNIX systems, implementations of threads that adhere to the IEEE POSIX 1003.1c standard are Pthreads.

pthreads

- For UNIX systems, implementations of threads that adhere to the IEEE POSIX 1003.1c standard are Pthreads.
- The primary motivation behind Pthreads is improving program performance
- Can be created with much less OS overhead & Needs fewer system resources to run.

pthread library

- Programs must include the file pthread.h
- Programs must be linked with the pthread library
  - gcc -lpthread main.c -o main

- Some functions
  - pthread_create(): Creates a new thread
  - pthread_exit():Terminates the calling thread
  - pthread_mutex_lock(): lock a mutex
  - pthread_join(): Causes the calling thread to wait for another thread to terminate

pthread library

- Programs must include the file pthread.h
- Programs must be linked with the pthread library
  - gcc -lpthread main.c -o main

- Some functions
  - pthread_create(): Creates a new thread
  - pthread_exit():Terminates the calling thread
  - pthread_mutex_lock(): lock a mutex
  - pthread_join(): Causes the calling thread to wait for another thread to terminate

explore the functions !!!

Example: Creating a thread with Pthreads


```c
#include <pthread.h>

void *thread_function(void *arg) {
    // Thread code here
    return NULL;
}

int main() {
    pthread_t thread;
    pthread_create(&thread, NULL, thread_function, NULL);
    pthread_join(thread, NULL);
    return 0;
}
```

C++ std::thread

- Provides a higher-level abstraction over native threads

C++ std::thread

- Provides a higher-level abstraction over native threads
- Automatic resource management
- Integrates well with other C++ features (lambdas, function objects)
- Easier to write clean code

C++ std::thread

- Provides a higher-level abstraction over native threads
- Automatic resource management
- Integrates well with other C++ features (lambdas, function objects)
- Easier to write clean code

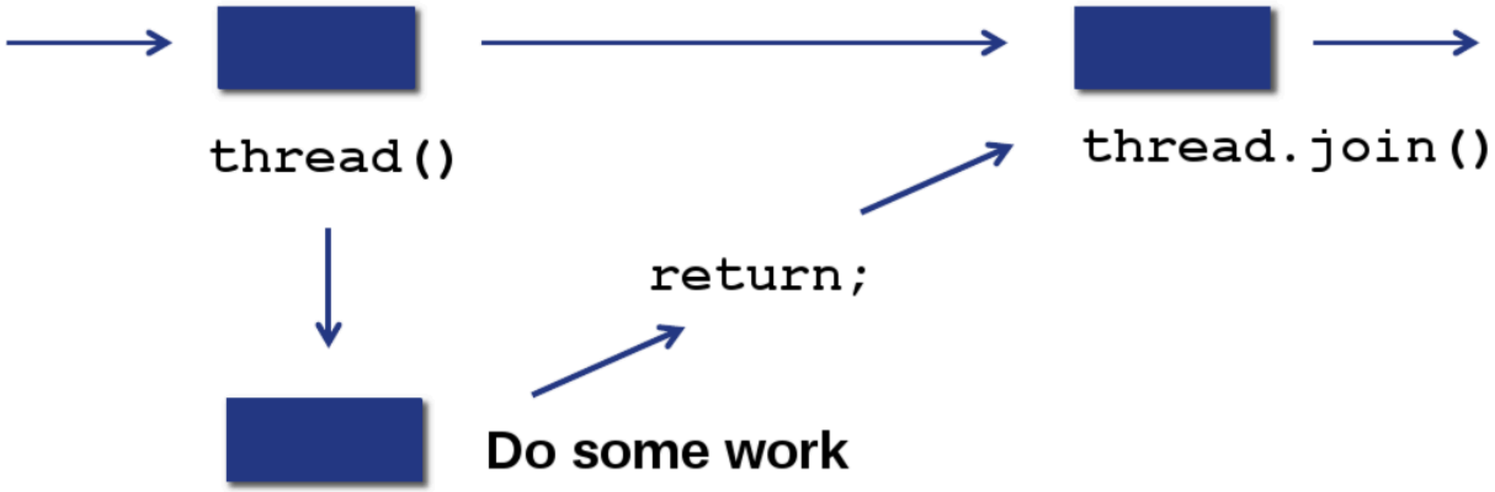
- Disadvantages:
  - Not all synchronization primitives are implemented, e.g., barriers (do not worry, you will not implement it this week).
  - A modern compiler is needed

Thread creation


```cpp
#include <thread>
#include <iostream>
#include <string>

void greet(const std::string& name) {
    std::cout << "Hello, " << name << "!" << std::endl;
}

int main() {
    std::thread t(greet, "Alice");
    t.join();
    return 0;
}
```

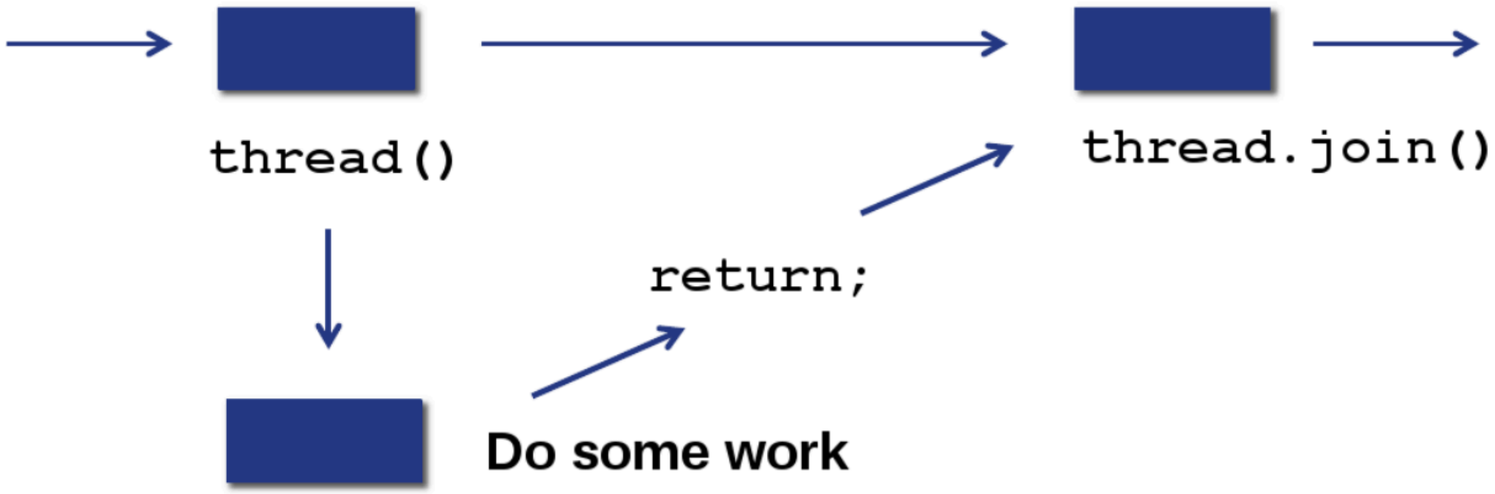
Joining of threads

- join(): Wait for thread to finish



Joining of threads

- join(): Wait for thread to finish




```
std::thread t(some_function);
t.join(); // Wait for t to finish
```

Can you find out the problem in the above implementation ?

Can you find out the problem in the above implementation ?

- Do you all remember about the catch in OS threads ?

  Latency & Overhead

Can you find out the problem in the above implementation ?

- Do you all remember about the catch in OS threads ?

  Latency & Overhead

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

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

Whats the solution ?

- We need to create infrastructure for task loading and unloading the the threads.

Whats the solution ?

- We need to create infrastructure for task loading and unloading the the threads.

- And thats called an “EVENT LOOP”

Whats the solution ?

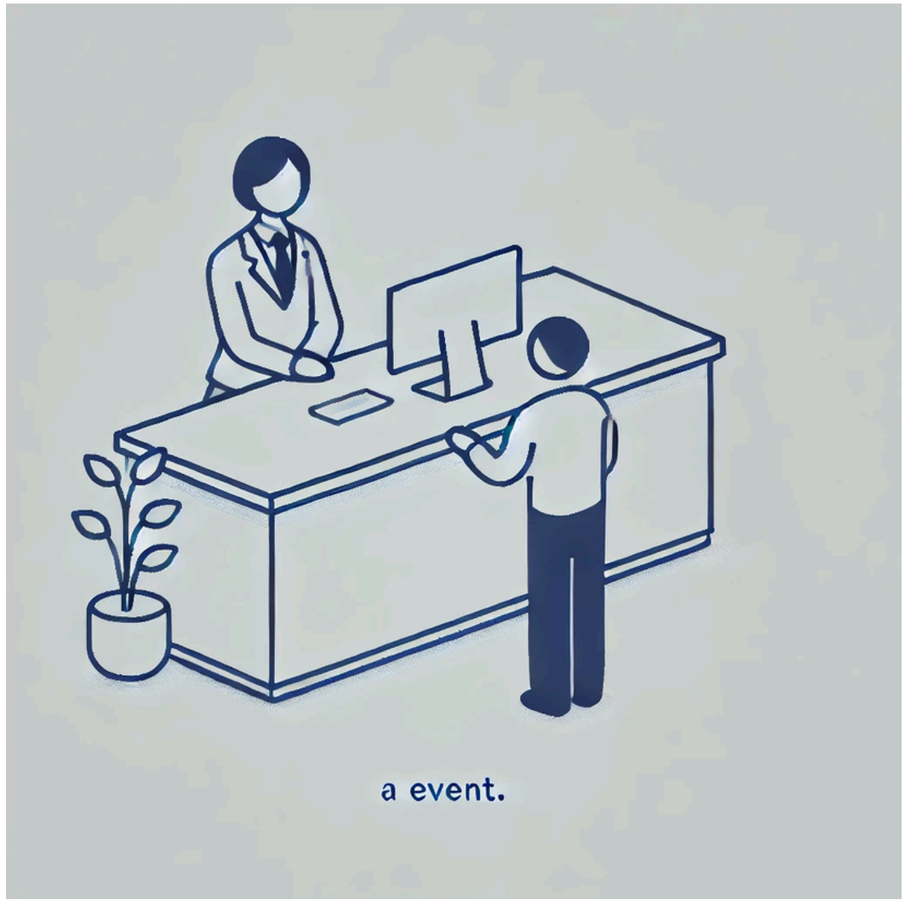
- We need to create infrastructure for task loading and unloading the the threads.

- And thats called an “EVENT LOOP”

- And what is an event ?
  - Events can be things like keyboard input, mouse clicks, or network data (like interrupts)

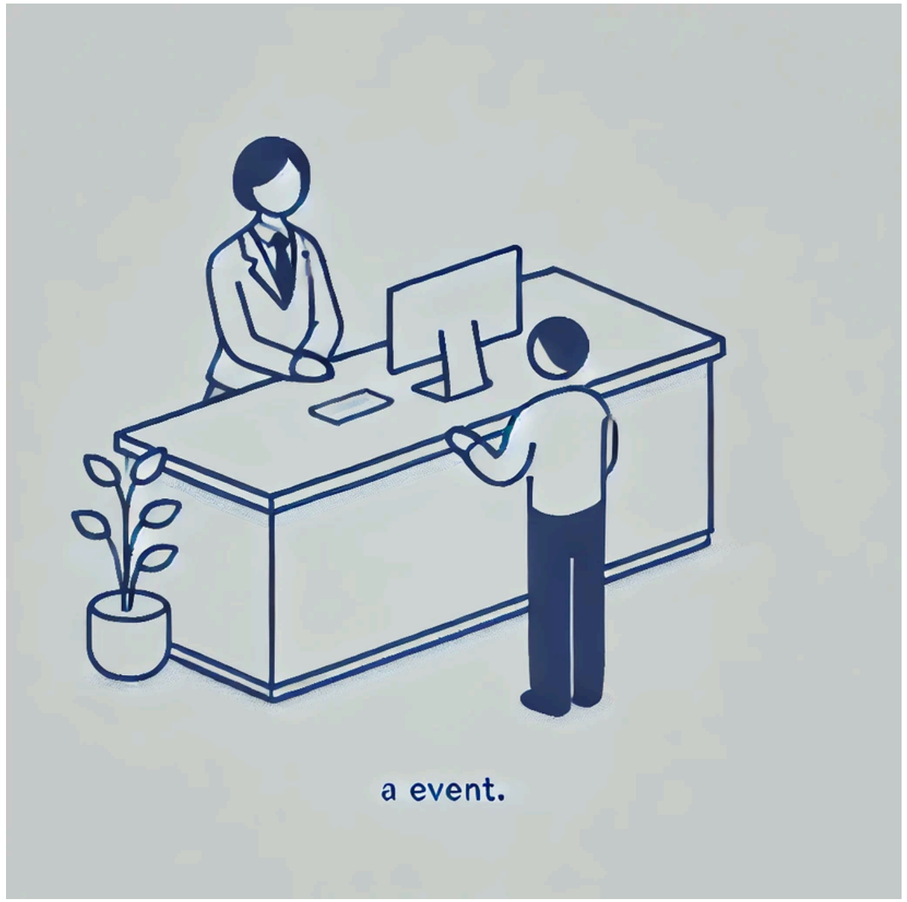
Analogy to an event loop

- The receptionist waits for someone (event) to walk in.



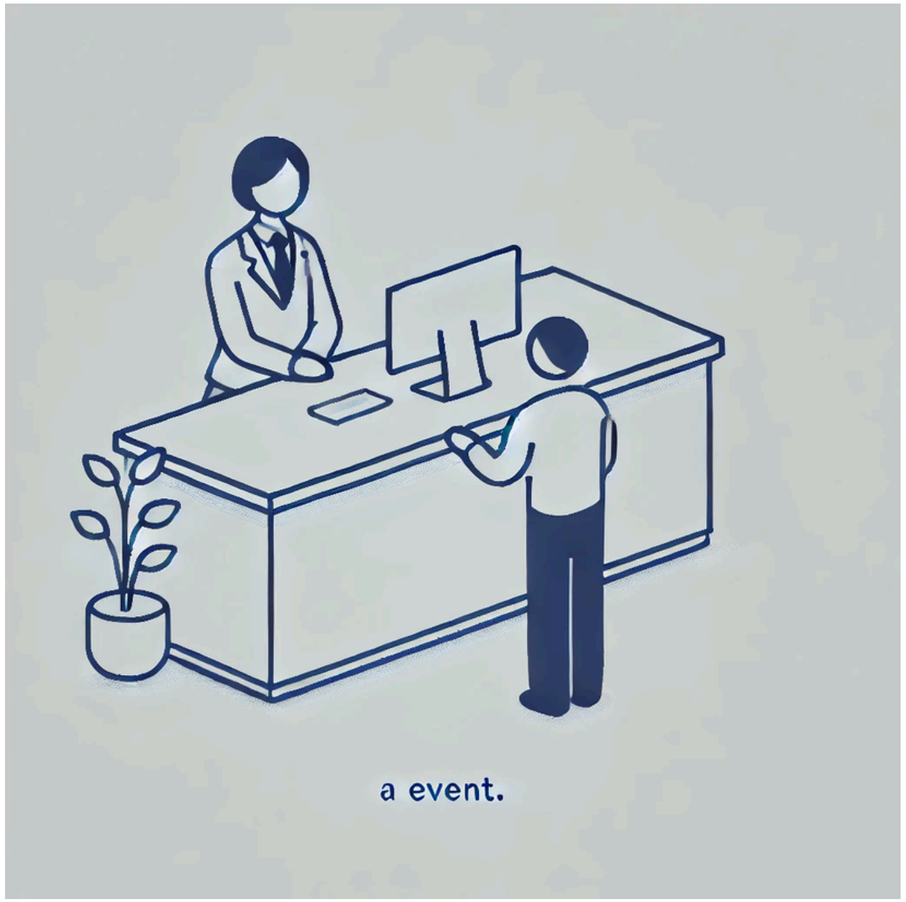
Analogy to an event loop

- The receptionist waits for someone (event) to walk in.
- When someone comes in with a question (event), the receptionist helps them (processes the event).



Analogy to an event loop

- The receptionist waits for someone (event) to walk in.
- When someone comes in with a question (event), the receptionist helps them (processes the event).
- Once that person is helped, the receptionist is ready for the next person.(event)



So what is an event loop ?

- An event loop in an operating system is like a manager that listens for events.

So what is an event loop ?

- An event loop in an operating system is like a manager that listens for events.

- When an event happens, the event loop processes it by triggering the appropriate action (function or handler).

So what is an event loop ?

- An event loop in an operating system is like a manager that listens for events.
- When an event happens, the event loop processes it by triggering the appropriate action (function or handler).
- After processing the event, the event loop goes back to listening for more events.

So what is an event loop ?

- An event loop in an operating system is like a manager that listens for events.
- When an event happens, the event loop processes it by triggering the appropriate action (function or handler).
- After processing the event, the event loop goes back to listening for more events.
- It allows the system to handle multiple tasks without getting stuck on one.

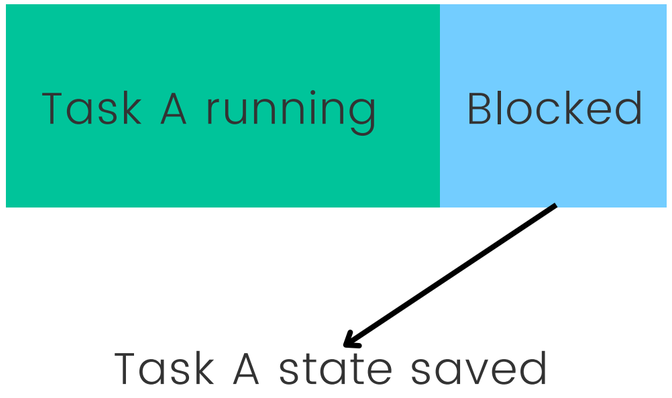
How to implement concurrency? Step-1

- Start a task A, and do required computation

Task A running

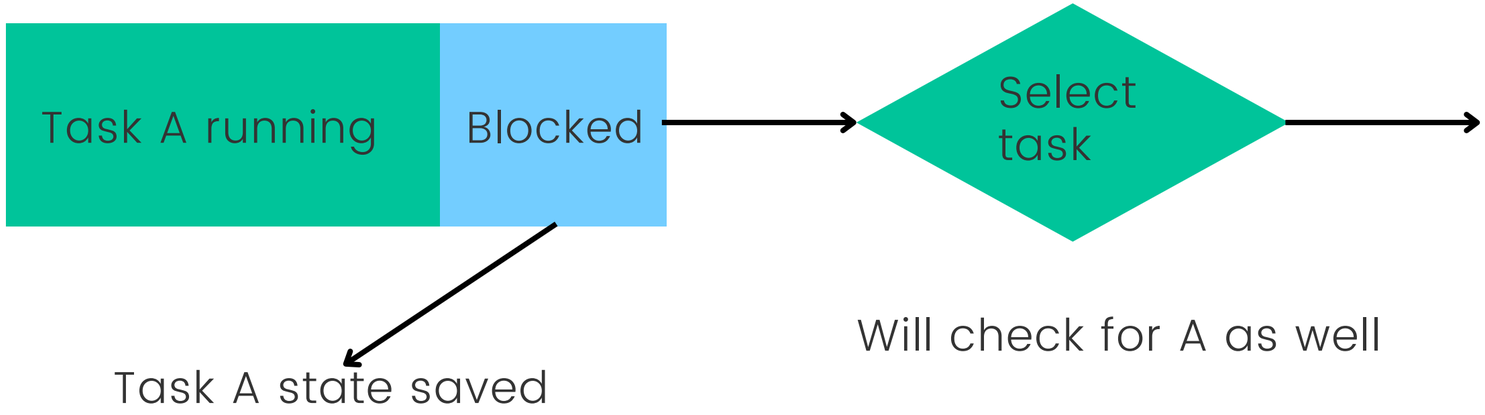
How to implement concurrency? Step-2

- Start a task A, and do required computation
- When task A blocks/waits, save state of A



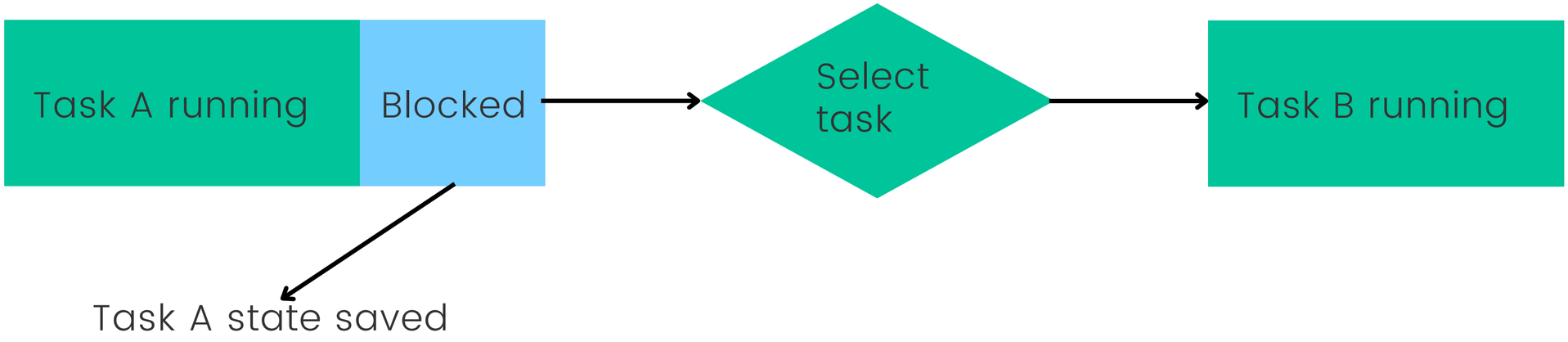
How to implement concurrency? Step-3

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



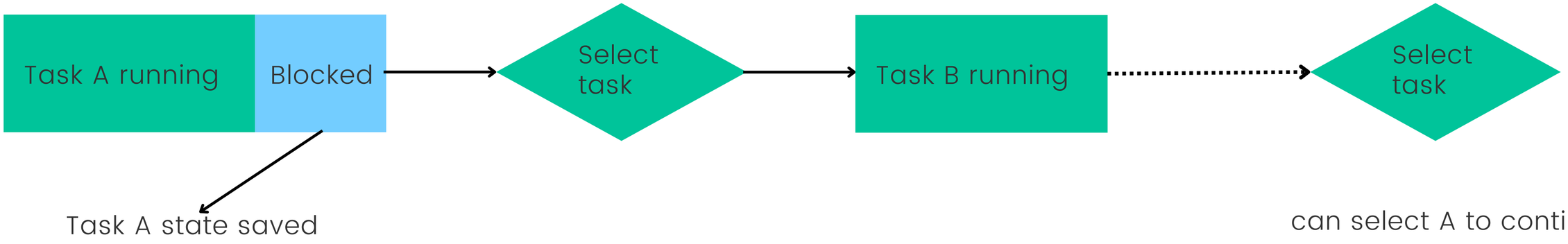
How to implement concurrency? Step-4 (1/2)

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



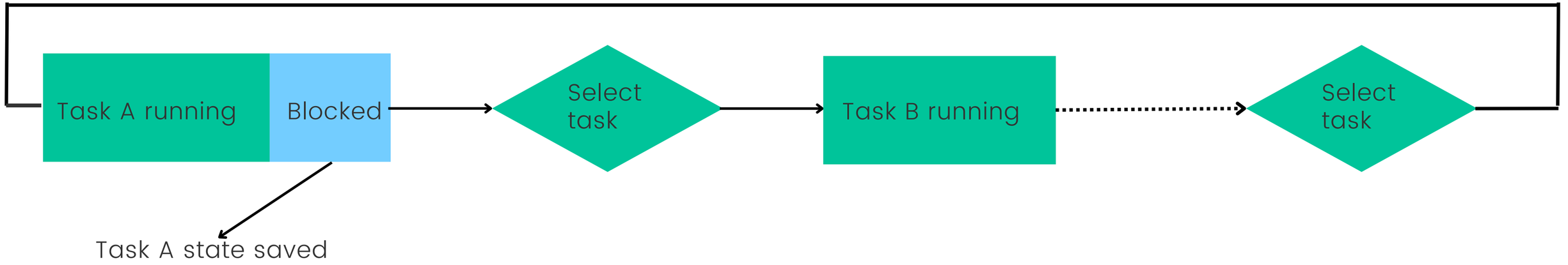
How to implement concurrency? Step-4 (2/2)

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



How to implement concurrency? Step-5

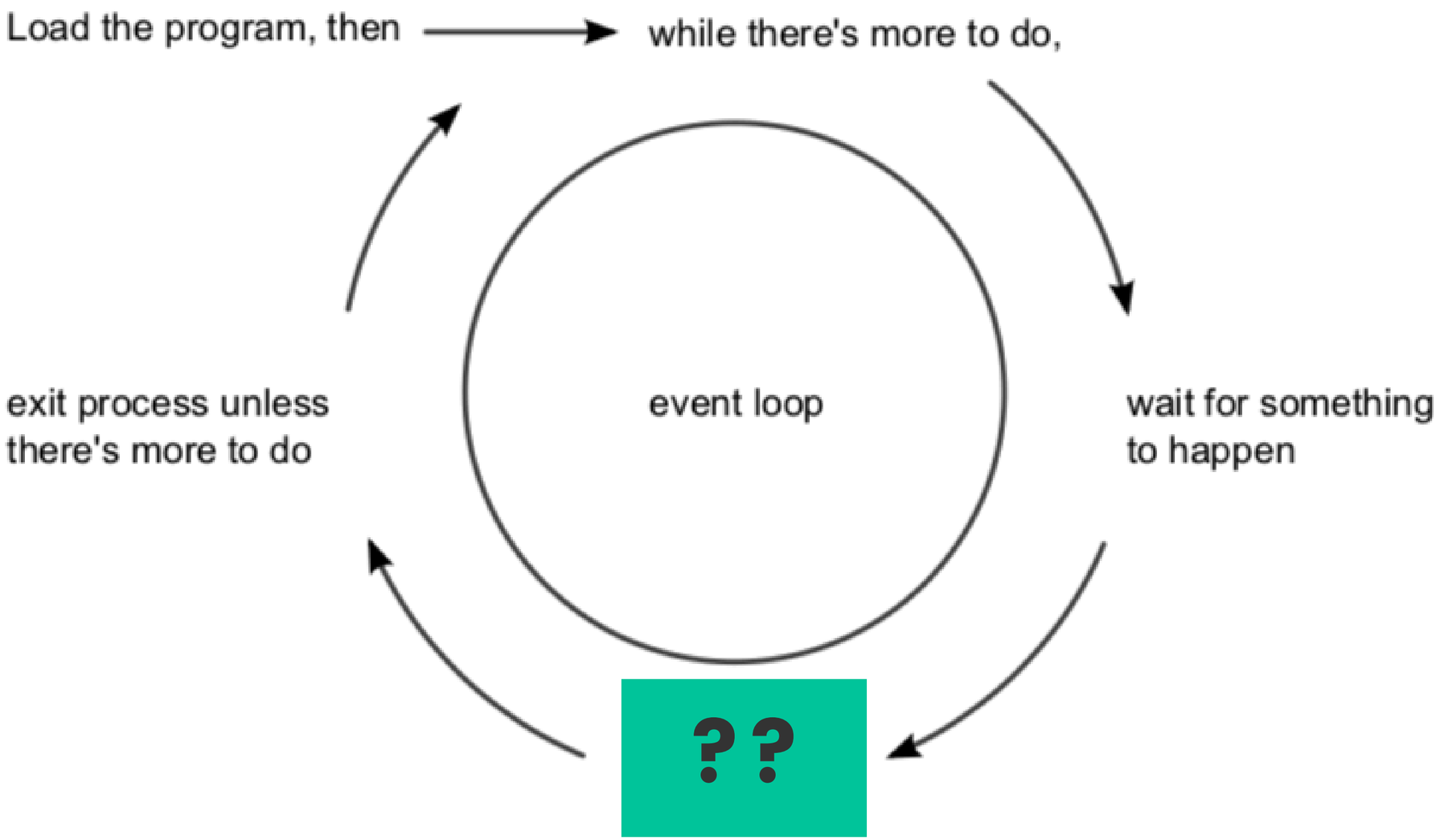
- Start a task A, and do required computation
- When task A blocks/waits, save state of A
- Add check in selector to see if A is unblocked, i.e. ready to run again
- Check for “ready tasks”, and select one among them to run
- Repeat for each task! (yep this is the event loop)



So, what’s next ?

- We have completed “unloading” a function.
- So Whats the next Step ?

Hint


Load the program, then
while there's more to do,
wait for something
to happen
exit process unless
there's more to do
event loop
??

So, what's next ?

- We have completed "unloading" a function.

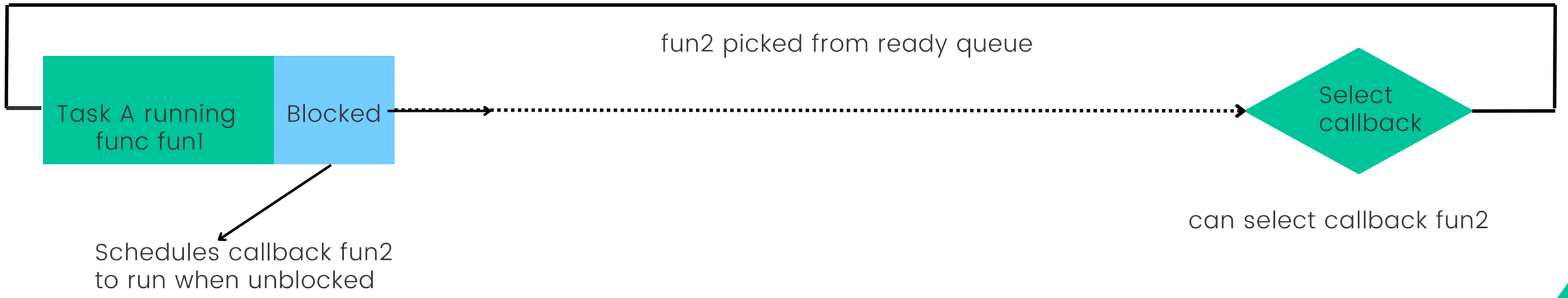
- We have to reload the functions ?

~ CALLBACKS

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 & callbacks

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

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

while true:

1. Select callback to run from ready queue
2. Run the callback
3. if needed : schedule a new callback

Event loops & callbacks

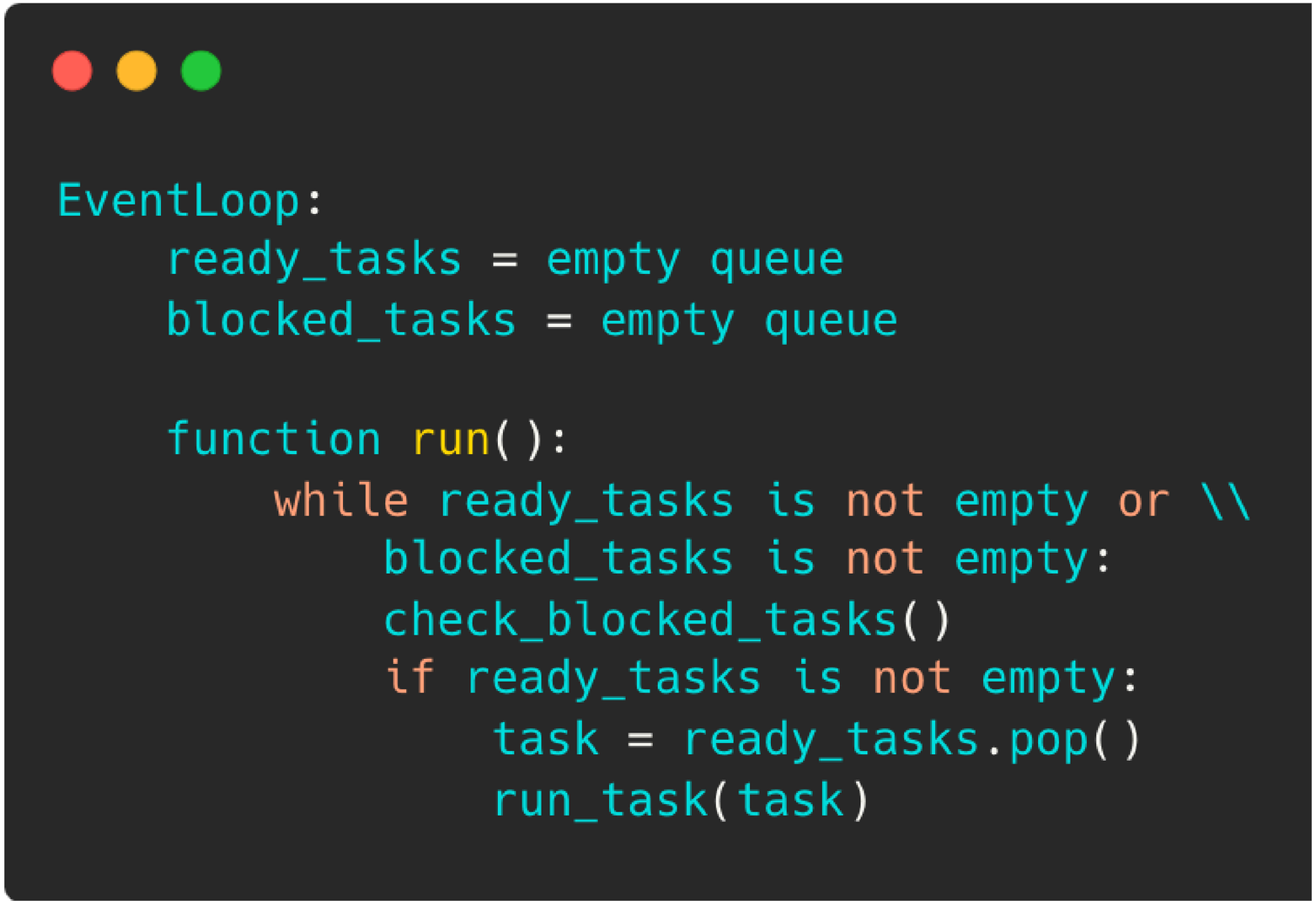
- Start a task A, and do required computation
- When task A blocks/waits, save state of A
- Add check in selector to see if A is unblocked, i.e. ready to run again
- Check for “ready tasks”, and select one among them to run
- Repeat for each task! (yep this is the event loop)

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

while true:

1. Select callback to run from ready queue
2. Run the callback
3. if needed : schedule a new callback

Let’s code it …



```
EventLoop:
    ready_tasks = empty queue
    blocked_tasks = empty queue

    function run():
        while ready_tasks is not empty or \\
            blocked_tasks is not empty:
            check_blocked_tasks()
            if ready_tasks is not empty:
                task = ready_tasks.pop()
                run_task(task)
```

Let's code it ...


```
EventLoop:
    ready_tasks = empty queue
    blocked_tasks = empty queue

    function run():
        while ready_tasks is not empty or \\
            blocked_tasks is not empty:
            check_blocked_tasks()
            if ready_tasks is not empty:
                task = ready_tasks.pop()
                run_task(task)
```

```
    function check_blocked_tasks():
        for each task in blocked_tasks:
            if task is ready:
                move task from blocked_tasks to ready_tasks
    function run_task(task):
        resume task
        if task is complete:
            clean up task
        else if task is blocked:
            move task to blocked_tasks
        else:
            add task back to ready_tasks
```

Let’s code it …

```
EventLoop:
    ready_tasks = empty queue
    blocked_tasks = empty queue

    function run():
        while ready_tasks is not empty or blocked_tasks is not empty:
            check_blocked_tasks()
            if ready_tasks is not empty:
                task = ready_tasks.pop()
                run_task(task)

    function check_blocked_tasks():
        for each task in blocked_tasks:
            if task is ready:
                move task from blocked_tasks to ready_tasks

    function run_task(task):
        resume task
        if task is complete:
            clean up task
        else if task is blocked:
            move task to blocked_tasks
        else:
            add task back to ready_tasks
```

```
Task:
    function resume():
        continue execution until next yield point

    function is_complete():
        return true if task has finished, false otherwise

    function is_blocked():
        return true if task is waiting for I/O, false otherwise

    function is_ready():
        return true if blocked task can now proceed, false otherwise

Main program:
    loop = new EventLoop
    add initial tasks to loop.ready_tasks
    loop.run()
```


CURRENT MOOD

Let's start the assignment !!!

CS69201: Computing Lab-1

Systems Assignment 1: Basic and Non-Blocking Execution Environment for C++ Threads

Deadline: October 14, 2024, 11:59 PM

Let's start the assignment !!!

CS69201: Computing Lab-1

Systems Assignment 1: Basic and Non-Blocking Execution Environment for C++ Threads

Deadline: October 14, 2024, 11:59 PM


When everyone is on holiday, but you have to work

We have another tutorial tomorrow :)

- Date : 2nd October , 2024 [Wednesday]
- Location : CSE 120
- Time : 2-3 PM

Thank you for listening!

2, October 2024

CSE, IIT KGP

Systems Programming: Threading (Part 2)

Computing Lab (CS 69201)

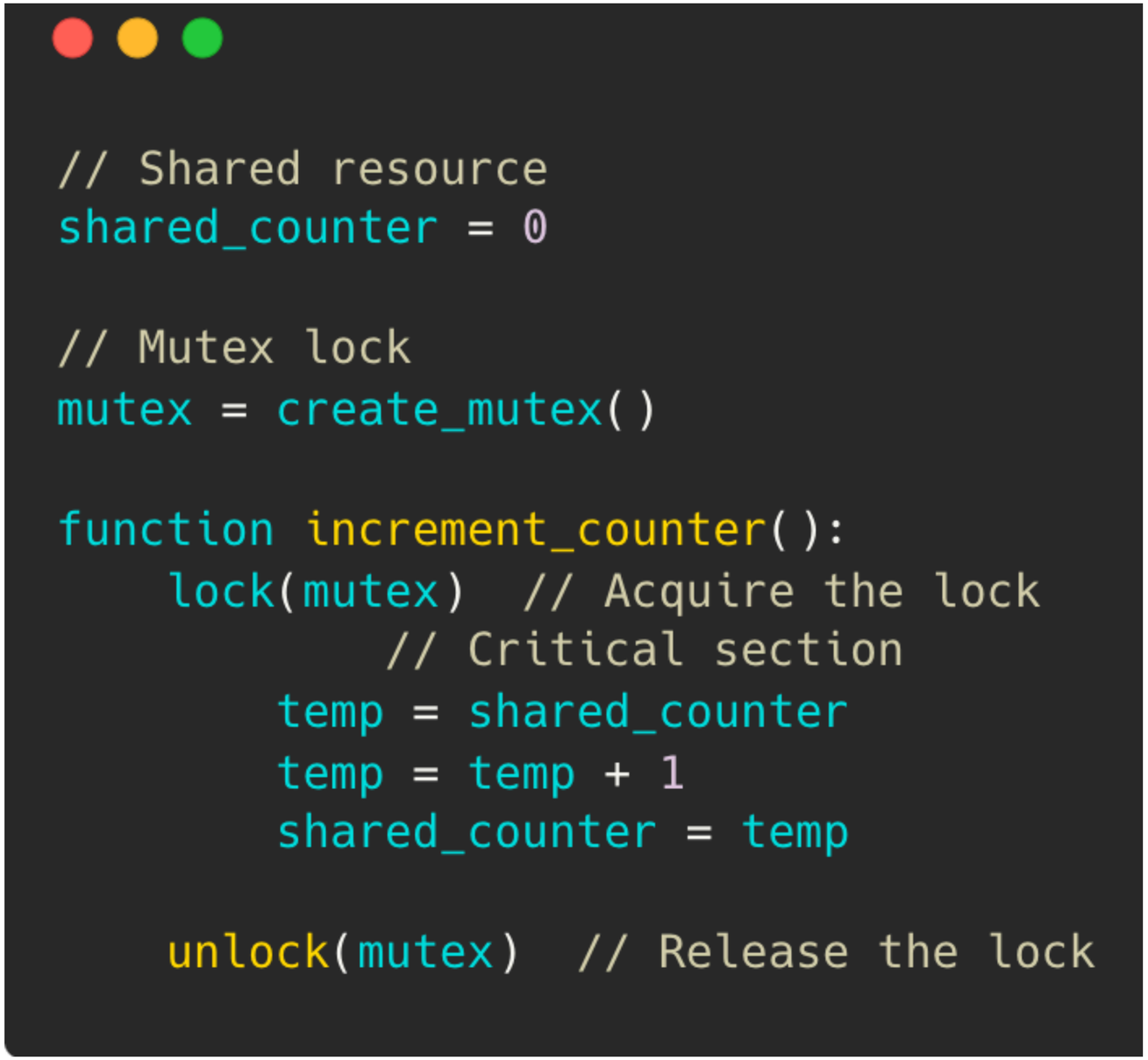
Department of Computer Science and Engineering

IIT Kharagpur

Synchronisation problem

- Do you all know about the critical section problem?
- To get away with it- we use locks.
- Locks :
  - Mutex Locks

Mutex Locks



```
// Shared resource
shared_counter = 0

// Mutex lock
mutex = create_mutex()

function increment_counter():
    lock(mutex)  // Acquire the lock
                 // Critical section
        temp = shared_counter
        temp = temp + 1
        shared_counter = temp

    unlock(mutex)  // Release the lock
```

Mutex Locks

```
// Shared resource
shared_counter = 0

// Mutex lock
mutex = create_mutex()

function increment_counter():
    lock(mutex)  // Acquire the lock
            // Critical section
        temp = shared_counter
        temp = temp + 1
        shared_counter = temp

    unlock(mutex)  // Release the lock
```

```
// Thread 1
function thread1_function():
    for i = 1 to 1000000:
        increment_counter()

// Thread 2
function thread2_function():
    for i = 1 to 1000000:
        increment_counter()

// Main program
create_thread(thread1_function)
create_thread(thread2_function)
wait_for_threads_to_finish()
print("Final counter value:", shared_counter)
```

Synchronisation problem

- Do you all know about the critical section problem?
- To get away with it- we use locks.
- Locks :
  - Mutex Locks
  - Semaphores

Semaphores


```
// Initialize a semaphore with 2 permits
semaphore = create_semaphore(2)

function process():
    wait(semaphore)  // P operation (down)

    // Critical section
    print("Entering critical section")
    // Simulate some work
    sleep(2 seconds)
    print("Exiting critical section")

    signal(semaphore)  // V operation (up)

// Main program
for i = 1 to 5:
    create_thread(process)

wait_for_all_threads_to_finish()
```

Synchronisation problem

- We all know about the critical section problem.
- To ensure atomicity - we use locks
- Locks :
  - Mutex Locks
  - Semaphores

- But what's the problem with locks?

Problem with locks

- Performance bottlenecks
- Mutex locks cause context switches
- ... more

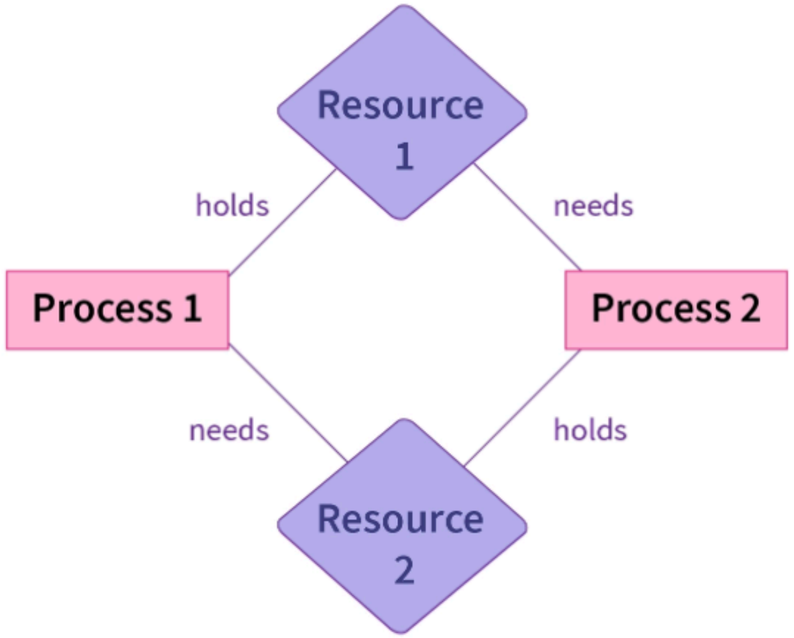


Image credits: scaler.com

Problem with locks

- Performance bottlenecks
- Mutex locks cause context switches
- ... more

So what's the way out ?

Lock-Free Programming

- Thread-safe access to shared data without the use of synchronization primitives such as mutexes.
- Possible but not practical in the absence of hardware support
- Designing them is hard.

Lock-Free Programming

- Thread-safe access to shared data without the use of synchronization primitives such as mutexes.
- Possible but not practical in the absence of hardware support
- Designing them is hard.

- Rather we design lock-free data structures.[stack, queue, buffer, map, deque etc.]

- But how ??

Lock-Free Programming

- Thread-safe access to shared data without the use of synchronization primitives such as mutexes.
- Possible but not practical in the absence of hardware support
- Designing them is hard.

- Rather we design lock-free data structures.[stack, queue, buffer, map, deque etc.]

- But how ??
  - Well, we will use lock-free primitives

Example (1/3)


```
Structure Node:
    data
    next (atomic access)
```

```
Class LockFreeQueue:
    head   // Points to the first node (for dequeue)
    tail   // Points to the last node (for enqueue)

    Constructor:
        Initialize a dummy node
        Set head and tail to the dummy node
```

Example (2/3)


```
Function Enqueue(value):
    new_node = new Node()
    new_node.data = value
    new_node.next = NULL

    Loop:
        current_tail = tail          // Load the current tail pointer
        tail_next = current_tail.next

        // Check if tail pointer is lagging behind (tail.next is not NULL)
        If tail_next != NULL:
            // Move tail forward to catch up
            Compare-And-Swap(tail, current_tail, tail_next)
        Else:
            // Try to append the new node to the end of the queue
            If Compare-And-Swap(current_tail.next, NULL, new_node):
                Break the loop   // Successfully linked the new node

    // Now move the tail pointer to the new node
    Compare-And-Swap(tail, current_tail, new_node)
```

Example (3/3)


```
Function Dequeue():
    Loop:
        current_head = head
        head_next = current_head.next

        // Check if the queue is empty (head.next is NULL)
        If head_next == NULL:
            Return NULL

        // Attempt to move the head forward
        If Compare-And-Swap(head, current_head, head_next):
            Return head_next.data    // Successfully dequeued the value
```

Lock-Free APIs in C++

- in C++ we have the atomic library for lock-free programming

- std::atomic provides various atomic data types like:
  - atomic_int
  - atomic_bool
  - atomic_char
  - atomic_intptr_t
  - …
- provides templatized access to atomic primitives
  - i.e. std::atomic<T>
  - can use with user defined data types (UDT) !!

Lock-Free APIs in C/C++ : Example


```cpp
#include <atomic>
std::atomic<int> counter(0);  // Atomic shared counter
void increment() {
    for (int i = 0; i<1000; i++)
        ++counter;  // Atomic increment (lock-free)
}

int main() {
    std::vector<std::thread> threads;
    for (int i = 0; i < 100; ++i) {
        threads.push_back(std::thread(increment));
    }
    for (auto& t : threads) {
        t.join();
    }
    std::cout << "Final counter value: " << counter << std::endl;
    return 0;
}
```

Advantages of Lock-Free Program

- No/Less Context-Switches
- Higher CPU frequency and throughput
- No Deadlocks or Priority Inversions
- Faster Multicore Programming

Problems with Lock-Free Program

- Usually, it is hard to write lock-free code
- It is even harder to write correct lock-free code
- Spin-Locks cause heavy memory usage
- Overall performance can go down in some cases compared to mutexes

Thank you for listening!