Міністерство освіти і науки України

Національний технічний університет України

“Київський політехнічний інститут імені Ігоря Сікорського”

Факультет інформатики та обчислювальної техніки

Кафедра інформатики та програмної інженерії

Комп’ютерний практикум №4

Моделювання систем

Тема: Оцінка точності та складності алгоритму імітації

Виконав
студент групи ІП-11:
Панченко С. В.

Перевірила:
Дифучина О. Ю.

Київ 2024

ЗМІСТ

1 Мета комп’ютерного практикуму.....6

2 Завдання.....7

3 Виконання.....8

3.1 Послідовна модель.....8

3.2 Теоретична оцінка складності.....11

3.3 Розгалужена модель.....12

3.4 Комплексна модель.....12

Висновок.....13

ДОДАТОК А ТЕКСТИ ПРОГРАМНОГО КОДУ.....14

1 МЕТА КОМП’ЮТЕРНОГО ПРАКТИКУМУ

Розробити та дослідити модель масового обслуговування, оцінити точність та складність алгоритму імітації експериментально та теоретично, а також проаналізувати вплив зміни структури мережі на результати.

2 ЗАВДАННЯ

1. Розробити модель масового обслуговування, яка складається з N систем масового обслуговування. Число N є параметром моделі. Кількість подій в моделі оцінюється числом N+1. 20 балів.
2. Виконати експериментальну оцінку складності алгоритму імітації мережі масового обслуговування. Для цього виконайте серію експериментів, в якій спостерігається збільшення часу обчислення алгоритму імітації при збільшенні кількості подій в моделі. 40 балів.
3. Виконати теоретичну оцінку складності побудованого алгоритму імітації. 30 балів.
4. Повторіть експеримент при зміні структури мережі масового обслуговування. 10 балів.

3 ВИКОНАННЯ

3.1 Послідовна модель

Побудуємо модель, яка складається з 50-ти послідовних процесів. Протестуємо її із часом симуляції від 10000 до 100000. Програма виконана мовою Rust 1.8.1. Тестування проводилося за відсутності сторонніх процесів у системі. Нижче зазначені характеристика комп’ютера та ОС.

Таблиця 3.1 Характеристики комп’ютера

system	HP 255 G8 Notebook PC (5N3M6EA#AKD)
bus	890E
memory	128KiB BIOS
processor	AMD Ryzen 5 5500U with Radeon Graphics
memory	384KiB L1 cache
memory	3MiB L2 cache
memory	8MiB L3 cache
memory	32GiB System Memory
memory	16GiB SODIMM DDR4 Synchronous Unbuffered (Unregistered) 2400 MHz (0,4 ns)
memory	16GiB SODIMM DDR4 Synchronous Unbuffered (Unregistered) 2400 MHz (0,4 ns)
bridge	Renoir/Cezanne Root Complex
generic	Renoir/Cezanne IOMMU
bridge	Renoir/Cezanne PCIe GPP Bridge
network	RTL8822CE 802.11ac PCIe Wireless Network Adapter
bridge	Renoir/Cezanne PCIe GPP Bridge
network	RTL8111/8168/8411 PCI Express Gigabit Ethernet Controller
bridge	Renoir/Cezanne PCIe GPP Bridge

storage	MTFDHBA512QFD-1AX1AABHA
disk	NVMe disk
disk	NVMe disk
disk	512GB NVMe disk
volume	511MiB Windows FAT volume
volume	476GiB EXT4 volume
bridge	Renoir Internal PCIe GPP Bridge to Bus
display	Lucienne
multimedia	Renoir Radeon High Definition Audio Controller
input	HD-Audio Generic HDMI/DP,pcm=3
generic	Family 17h (Models 10h-1fh) Platform Security Processor
bus	Renoir/Cezanne USB 3.1
bus	xHCI Host Controller
storage	Flash Card Reader/Writer
disk	31GB Card Reader
disk	31GB
bus	xHCI Host Controller
bus	Renoir/Cezanne USB 3.1
bus	xHCI Host Controller
communication	Bluetooth Radio
input	Logitech USB Optical Mouse
multimedia	HP TrueVision HD Camera: HP Tru
bus	xHCI Host Controller
multimedia	Raven/Raven2/FireFlight/Renoir Audio Processor
multimedia	Family 17h (Models 10h-1fh) HD Audio

	Controller
input	HD-Audio Generic Mic
input	HD-Audio Generic Headphone
bridge	Renoir Internal PCIe GPP Bridge to Bus
storage	FCH SATA Controller [AHCI mode]
storage	FCH SATA Controller [AHCI mode]
bus	FCH SMBus Controller
bridge	FCH LPC Bridge
system	PnP device PNP0c02
generic	PnP device HPQ8001
system	PnP device PNP0c02
system	PnP device PNP0c01
bridge	Renoir PCIe Dummy Host Bridge
bridge	Renoir PCIe Dummy Host Bridge
bridge	Renoir PCIe Dummy Host Bridge
bridge	Renoir Device 24: Function 0
bridge	Renoir Device 24: Function 1
bridge	Renoir Device 24: Function 2
bridge	Renoir Device 24: Function 3
bridge	Renoir Device 24: Function 4
bridge	Renoir Device 24: Function 5
bridge	Renoir Device 24: Function 6
bridge	Renoir Device 24: Function 7
power	HW03041
input	Power Button
input	Lid Switch
input	HP WMI hotkeys
input	ELAN072B:00 04F3:31BF Mouse

input	ELAN072B:00 04F3:31BF Touchpad
input	Power Button
input	AT Translated Set 2 keyboard
input	Video Bus
input	Wireless hotkeys

На рисунку 3.1 розглянемо залежність реального часу виконання до часу симуляції.

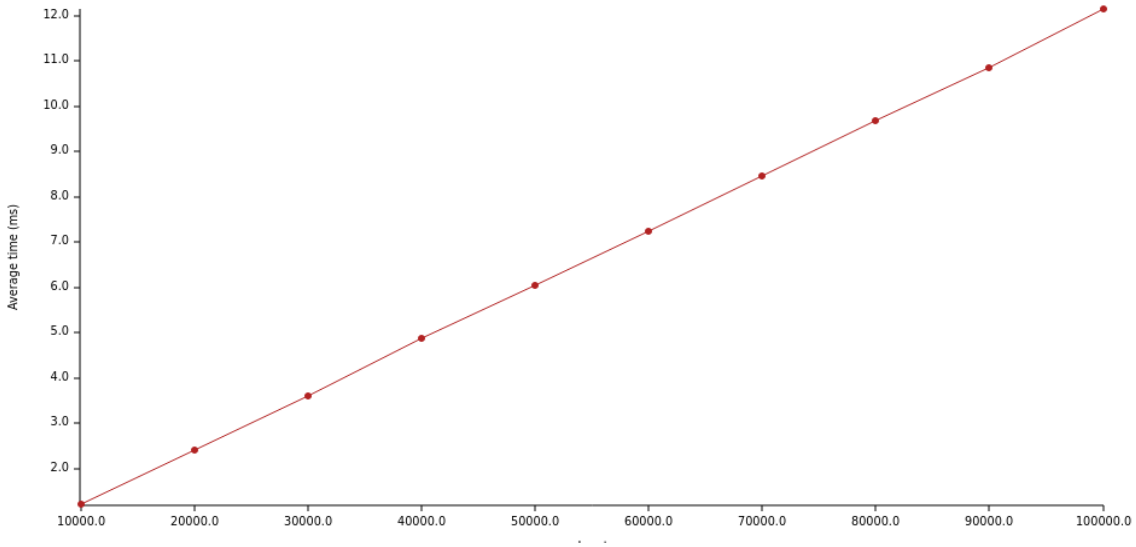

Рисунок 3.1 — Графік залежності реального часу виконання до часу симуляції для послідовної моделі

3.2 Теоретична оцінка складності

Теоретичну складність алгоритму можна виразити як O(λ * T * ω), де:

λ - частота виникнення подій (для генератора подій встановлено 1),

T - загальний час симуляції,

ω - усереднена кількість базових операцій для обробки однієї події (приймаємо за 50, враховуючи кількість процесорів).

Спрощуючи вираз, отримуємо O(1 * T * N) = O(N), що вказує на лінійну залежність складності від кількості подій. Експериментальні дані, відображені на графіку, підтверджують цю теоретичну оцінку, демонструючи лінійне зростання складності.

3.3 Розгалужена модель

На рисунку 3.2 розглянемо графік для розгалуженої моделі. Модель складається з 10 процесів. У кожному процесі ще по два дочірні процеси.

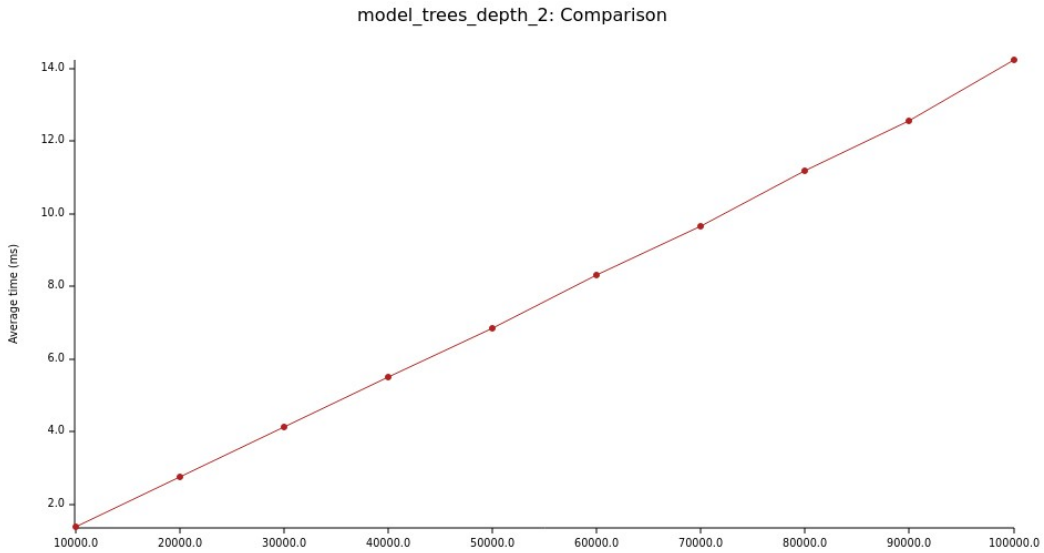

Рисунок 3.2 — Графік залежності реального часу виконання до часу симуляції для розгалуженої моделі

Бачимо, що залежність також лінійна.

3.4 Комплексна модель

Наразі нехай маємо 10 процесів, у кожному з них ще по два дочірні процесі. Кожен дочірній процес це 50 послідовних процесів. На рисунку 3.3 розглянемо графік залежності.

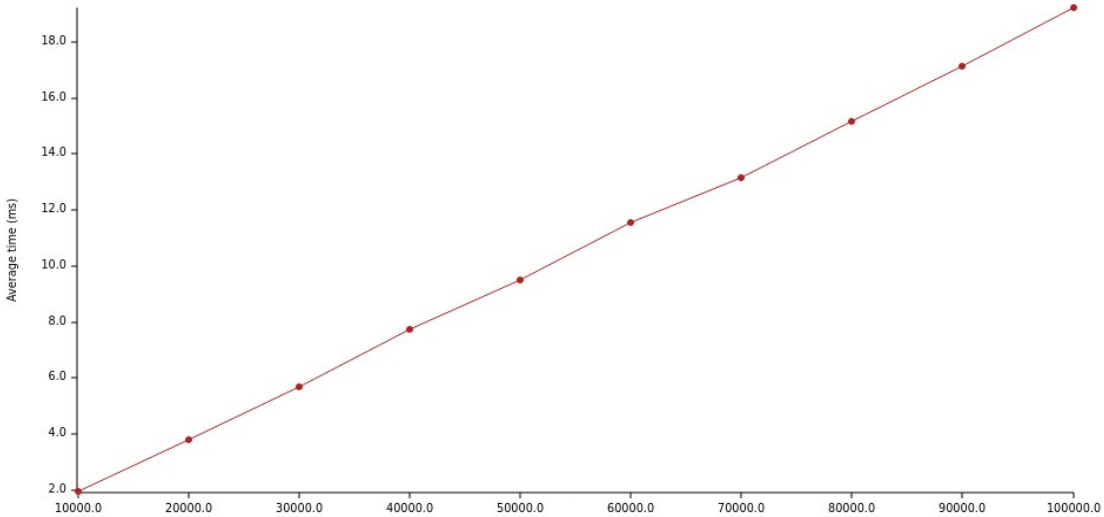

Рисунок 3.3— Графік залежності реального часу виконання до часу симуляції для комплексної моделі

Бачимо, що залежність також лінійна.

ВИСНОВОК

У ході виконання лабораторної роботи було розроблено та досліджено модель масового обслуговування, проведено оцінку точності та складності алгоритму імітації. На основі отриманих результатів можна зробити наступні висновки:

Експериментальна оцінка складності алгоритму імітації мережі масового обслуговування підтвердила теоретичні припущення щодо лінійної залежності часу виконання від кількості подій (ітерацій) в моделі. Це спостерігалось для всіх досліджених конфігурацій мережі.

Зміна структури мережі масового обслуговування впливає на абсолютні значення часу виконання, але не змінює характер залежності, яка залишається лінійною. Це підтверджує робастність алгоритму імітації до різних конфігурацій мережі.

Ускладнення структури мережі (збільшення кількості процесів та рівнів вкладеності) призводить до збільшення нахилу лінії на графіку, що відображає зростання часу виконання при тій самій кількості ітерацій.

Теоретична оцінка складності O(N), де N - кількість подій, добре узгоджується з експериментальними даними для всіх розглянутих конфігурацій мережі.

Розроблений алгоритм імітації демонструє передбачувану поведінку та масштабованість, що дозволяє ефективно застосовувати його для моделювання різноманітних систем масового обслуговування.

Отже, проведене дослідження підтвердило ефективність та точність розробленого алгоритму імітації мережі масового обслуговування, а також продемонструвало його гнучкість при моделюванні систем різної складності та структури.

ДОДАТОК А ТЕКСТИ ПРОГРАМНОГО КОДУ

Тексти програмного коду

(Найменування програми (документа))

Жорсткий диск

(Вид носія даних)

(Обсяг програми (документа), арк.)

Студента групи ІП-11 4 курсу

Панченка С. В

```
use std::cell::RefCell;
use std::cmp::Ordering;
use std::ops::{Add, Sub};
use crate::delay_gen::DelayGen;
use crate::event_create::EventCreate;
use crate::event_process::EventProcess;
use crate::node_create::NodeCreate;
use crate::node_process::NodeProcess;
use crate::payload_queue::PayloadQueue;
use crate::prob_arr::ProbabilityArray;

#[derive(Debug, Copy, Clone, Default, PartialOrd, PartialEq)]
pub struct TimePoint(pub f64);

impl Sub for TimePoint {
    type Output = TimeSpan;
    fn sub(self, rhs: TimePoint) -> TimeSpan {
        TimeSpan(self.0 - rhs.0)
    }
}

impl Add<TimeSpan> for TimePoint {
    type Output = TimePoint;
    fn add(self, rhs: TimeSpan) -> TimePoint {
        TimePoint(self.0 + rhs.0)
    }
}

#[derive(Debug, Copy, Clone, Default)]
pub struct TimeSpan(f64);

pub mod delay_gen {
    use rand::distributions::{Distribution};
    use crate::TimeSpan;
    use rand::thread_rng;
    use rand_distr::{Exp, Normal, Uniform};

    #[derive(Clone, Copy, Debug)]
    pub enum DelayGen {
        Normal(Normal<f64>),
        Uniform(Uniform<f64>),
        Exponential(Exp<f64>),
    }
```

```
    impl Default for DelayGen {
        fn default() -> DelayGen {
            DelayGen::Normal(Normal::new(0.0, 1.0).unwrap())
        }
    }

    impl DelayGen {
        pub fn sample(&self) -> TimeSpan {
            TimeSpan(
                match self {
                    Self::Normal(dist) => dist.sample(&mut thread_rng()),
                    Self::Uniform(dist) => dist.sample(&mut thread_rng()),
                    Self::Exponential(dist) => dist.sample(&mut thread_rng()),
                }
            )
        }
    }
}

pub mod queue_resource {
    use std::cell::{Cell};
    use std::rc::{Rc, Weak};

    pub trait Queue {
        type Item;
        fn push(&mut self, t: Self::Item);
        fn pop(&mut self) -> Option<Self::Item>;
        fn is_empty(&self) -> bool;
    }

    #[derive(Debug, Default, Clone)]
    pub struct QueueResource<Q> {
        max_acquires: usize,
        acquires_count: Rc<Cell<usize>>,
        queue: Q,
    }

    #[derive(Clone)]
    pub struct QueueProcessor<E> {
        acquires_count: Weak<Cell<usize>>,
        value: E
    }

    impl<E> Drop for QueueProcessor<E> {
```

```
        fn drop(&mut self) {
            let acquires_count = self.acquires_count.upgrade().expect("Queue
resource does not exist");
            acquires_count.set(acquires_count.get() - 1);
        }
    }

    impl<E> QueueProcessor<E> {
        // pub fn value(&self) -> &E {
        //     &self.value
        // }
        pub fn value_mut(&mut self) -> &mut E {
            &mut self.value
        }
    }

    impl<Q: Queue> QueueResource<Q> {
        pub fn new(queue: Q, max_acquires: usize) -> Self {
            Self{max_acquires, acquires_count: Rc::new(Cell::new(0usize)),
queue}
        }

        pub fn push(&mut self, t: Q::Item) {
            self.queue.push(t)
        }

        pub fn is_empty(&self) -> bool {
            self.queue.is_empty()
        }

        pub fn is_any_free_processor(&self) -> bool {
            self.acquires_count.get() < self.max_acquires
        }

        pub fn acquire_processor(&mut self) -> QueueProcessor<Q::Item> {
            assert!(self.acquires_count.get() < self.max_acquires);
            let value = self.queue.pop().expect("Queue is empty");
            self.acquires_count.set(self.acquires_count.get() + 1);
            QueueProcessor{acquires_count: Rc::downgrade(&self.acquires_count),
value}
        }
    }

    #[cfg(test)]
```

```
mod tests {
    use super::{Queue, QueueResource};

    #[derive(Default)]
    struct DummyQueue {
        len: usize
    }

    impl Queue for DummyQueue {
        type Item = ();
        fn push(&mut self, _: ()) {
            self.len += 1;
        }

        fn pop(&mut self) -> Option<()> {
            assert_eq!(self.is_empty(), false);
            self.len -= 1;
            Some(())
        }

        fn is_empty(&self) -> bool {
            self.len == 0
        }
    }

    #[test]
    fn test_one() {
        let mut res = QueueResource::new(DummyQueue::default(), 3usize);
        res.push(());
        res.push(());
        res.push(());
        res.push(());
        let _proc_one = res.acquire_processor();
        let _proc_two = res.acquire_processor();
        let _proc_three = res.acquire_processor();
    }

    #[test]
    #[should_panic]
    fn test_two() {
        let mut res = QueueResource::new(DummyQueue::default(), 2usize);
        res.push(());
        res.push(());
        res.push(());
```

```
            res.push(());

            let _proc_one = res.acquire_processor();
            let _proc_two = res.acquire_processor();
            let _proc_three = res.acquire_processor();
        }

        #[test]
        fn test_three() {
            let mut res = QueueResource::new(DummyQueue::default(), 2usize);
            res.push(());
            res.push(());
            res.push(());
            res.push(());

            let _proc_one = res.acquire_processor();
            let _proc_two = res.acquire_processor();
            drop(_proc_two);
            let _proc_three = res.acquire_processor();
        }

    }
}

pub mod prob_arr {
    use rand::random;

    #[derive(Default, Clone, Copy)]
    pub struct Probability(f64);
    impl Probability {
        pub fn new(prob: f64) -> Self {
            assert!(prob >= 0.0 && prob <= 1.0);
            Self(prob)
        }
    }

    #[derive(Default, Clone)]
    pub struct ProbabilityArray<T>(Vec<(T, Probability)>);

    impl<T> ProbabilityArray<T> {
        pub fn new(next_elements_map: Vec<(T, Probability)>) -> Self {
            const EPSILON: f64 = 0.001;
            let sum = next_elements_map.iter().map(|e| (*e).1.0).sum::<f64>();
            assert!((sum - 1.0).abs() < EPSILON);
```

```
            Self(next_elements_map)
        }

        pub fn sample(&self) -> Option<&T> {
            if self.0.is_empty() {
                return None;
            }

            let rand_value = random::<f64>();
            let mut current_sum = 0.0;

            let mut target_index = self.0.len() - 1;
            for (index, (_, prob)) in self.0.iter().enumerate() {
                current_sum += prob.0;
                if rand_value < current_sum {
                    target_index = index;
                    break;
                }
            }
            Some(&self.0.iter().nth(target_index)?.0)
        }
    }
}

#[derive(Clone, Default)]
pub struct Payload();

pub mod event_create {
    use crate::node_create::NodeCreate;
    use crate::{Event, Node, Payload, TimePoint};
    use crate::node_process::NodeProcess;

    #[derive(Clone)]
    pub struct EventCreate {
        current_t: TimePoint,
        node: *const NodeCreate
    }

    impl EventCreate {
        pub fn new(current_t: TimePoint, node: *const NodeCreate) -> Self {
            Self{current_t, node}
        }

        pub fn get_current_t(&self) -> TimePoint {
```

```
            self.current_t
        }

        pub fn iterate(self) -> (Self, Option<Event>) {
            let node = unsafe { &*self.node };
            let next_event = if let Some(next_node) = node.next_node() {
                match next_node {
                    Node::Create(node_create) => {

Some(Event::Create(node_create.produce_event(self.current_t)))
                    },
                    Node::Process(node_process) => {
                        NodeProcess::push_produce_event(node_process,
self.current_t, Payload()).map(Event::Process)
                    }
                }
            } else {
                None
            };
            (node.produce_event(self.current_t), next_event)
        }
    }
}

pub mod event_process {
    use std::cell::RefCell;
    use crate::{Event, Node, Payload, TimePoint};
    use crate::node_process::NodeProcess;
    use crate::queue_resource::QueueProcessor;

    #[derive(Clone)]
    pub struct EventProcess {
        current_t: TimePoint,
        node: *const RefCell<NodeProcess>,
        queue_processor: QueueProcessor<Payload>
    }

    impl EventProcess {
        pub fn new(
            current_t: TimePoint,
            node: *const RefCell<NodeProcess>,
            queue_processor: QueueProcessor<Payload>
        ) -> Self {
            Self{current_t, node, queue_processor}
```

```
        }

        pub fn get_current_t(&self) -> TimePoint {
            self.current_t
        }

        pub fn iterate(mut self) -> (Option<Self>, Option<Event>) {
            let node = unsafe { &*self.node };
            let payload = std::mem::take(self.queue_processor.value_mut());
            drop(self.queue_processor);

            let next_event = {
                let node = node.borrow();
                if let Some(next_node) = node.next_node() {
                    match next_node {
                        Node::Create(node_create) => {

Some(Event::Create(node_create.produce_event(self.current_t)))
                        },
                        Node::Process(node_process) => {
                            NodeProcess::push_produce_event(node_process,
self.current_t, payload).map(Event::Process)
                        }
                    }
                } else {
                    None
                }
            };
            (NodeProcess::pop_produce_event(node, self.current_t), next_event)
        }
    }
}

#[derive(Clone)]
pub enum Event {
    Create(EventCreate),
    Process(EventProcess),
}

impl Event {
    pub fn get_current_t(&self) -> TimePoint {
        match self {
            Event::Create(event) => event.get_current_t(),
            Event::Process(event) => event.get_current_t(),
```

```
        }
    }
}

impl Eq for Event {}

impl PartialEq<Self> for Event {
    fn eq(&self, other: &Self) -> bool {
        self.get_current_t() == other.get_current_t()
    }
}

impl PartialOrd<Self> for Event {
    fn partial_cmp(&self, other: &Self) -> Option<Ordering> {
        other.get_current_t().partial_cmp(&self.get_current_t())
    }
}

impl Ord for Event {
    fn cmp(&self, other: &Self) -> Ordering {
        self.partial_cmp(other).expect("Cannot compare events")
    }
}

#[derive(Default, Clone)]
pub struct NodeBase {
    next_nodes: ProbabilityArray<Node>,
    delay_gen: DelayGen
}

impl NodeBase {
    pub fn new(next_nodes: ProbabilityArray<Node>, delay_gen: DelayGen) -> Self
{
        Self {next_nodes, delay_gen}
    }
}

pub mod node_create {
    use crate::{EventCreate, Node, NodeBase, TimePoint};

    #[derive(Default, Clone)]
    pub struct NodeCreate(NodeBase);

    impl NodeCreate {
```

```
        pub fn new(base: NodeBase) -> Self {
            Self(base)
        }

        pub fn next_node(&self) -> Option<&Node> {
            self.0.next_nodes.sample()
        }

        pub fn produce_event(&self, old_t: TimePoint) -> EventCreate {
            EventCreate::new(old_t + self.0.delay_gen.sample(), self)
        }
    }
}

pub mod payload_queue {
    use crate::Payload;
    use crate::queue_resource::Queue;

    #[derive(Default, Clone)]
    pub struct PayloadQueue{
        len: usize,
    }

    impl Queue for PayloadQueue {
        type Item = Payload;

        fn push(&mut self, _: Self::Item) {
            self.len += 1;
        }

        fn pop(&mut self) -> Option<Self::Item> {
            if self.is_empty() {
                None
            } else {
                self.len -= 1;
                Some(Payload())
            }
        }

        fn is_empty(&self) -> bool {
            self.len == 0
        }
    }
}
```

```
pub mod node_process {
    use std::cell::RefCell;
    use crate::{EventProcess, Node, NodeBase, Payload, PayloadQueue, TimePoint};
    use crate::queue_resource::QueueResource;

    #[derive(Clone)]
    pub struct NodeProcess {
        base: NodeBase,
        queue: QueueResource<PayloadQueue>
    }

    impl NodeProcess {
        pub fn new(base: NodeBase, queue: QueueResource<PayloadQueue>) -> Self {
            Self { base, queue }
        }

        pub fn next_node(&self) -> Option<&Node> {
            self.base.next_nodes.sample()
        }

        pub fn pop_produce_event(node: &RefCell<Self>, old_t: TimePoint) ->
Option<EventProcess> {
            let is_any_free_processor =
node.borrow().queue.is_any_free_processor();
            let is_not_empty = !node.borrow().queue.is_empty();
            if is_any_free_processor && is_not_empty {
                let delay = node.borrow().base.delay_gen.sample();
                Some(EventProcess::new(old_t + delay, node,
node.borrow_mut().queue.acquire_processor()))
            } else {
                None
            }
        }

        pub fn push_produce_event(node: &RefCell<Self>, old_t: TimePoint,
payload: Payload) -> Option<EventProcess> {
            node.borrow_mut().queue.push(payload);
            NodeProcess::pop_produce_event(node, old_t)
        }
    }
}

#[derive(Clone)]
```

```
pub enum Node {
    Create(NodeCreate),
    Process(RefCell<NodeProcess>),
}

impl Default for Node {
    fn default() -> Self {
        Node::Create(Default::default())
    }
}

#[cfg(test)]
mod tests {
    use std::collections::BinaryHeap;
    use rand::distributions::Uniform;
    use crate::prob_arr::Probability;
    use crate::queue_resource::QueueResource;
    use super::*;

    #[test]
    fn test_one() {
        let tree = NodeCreate::new(
            NodeBase::new(
                ProbabilityArray::<Node>::new(
                    vec![
                        (
                            Node::Process(RefCell::new(NodeProcess::new(
                                NodeBase::new(
                                    Default::default(),
                                    DelayGen::Uniform(Uniform::new(5.0, 15.0))
                                ),
                                QueueResource::new(
                                    PayloadQueue::default(),
                                    3
                                )
                            ))),
                            Probability::new(1.0)
                        ),
                    ]
                ),
                DelayGen::Uniform(Uniform::new(1.0, 3.0))
            )
        );
```

```
    let mut events = BinaryHeap::<Event>::new();
    events.push(Event::Create(tree.produce_event(TimePoint(0.0))));

    let end_time = TimePoint(100000.0);
    let _ = loop {
        let next_event = events.pop().unwrap();
        if next_event.get_current_t() > end_time {
            break next_event;
        }

        match next_event {
            Event::Create(event) => {
                let (event_self, next_event) = event.iterate();
                events.push(Event::Create(event_self));
                if let Some(next_event) = next_event {
                    events.push(next_event)
                }
            },
            Event::Process(event) => {
                let (event_self, next_event) = event.iterate();
                if let Some(event_self) = event_self {
                    events.push(Event::Process(event_self))
                }
                if let Some(next_event) = next_event {
                    events.push(next_event)
                }
            }
        }
    };
}

#[test]
fn test_two() {
    let prob_array = {
        let prob_array = {
            let node_prob = (
                Node::Process(RefCell::new(NodeProcess::new(
                    NodeBase::new(
                        Default::default(),
                        DelayGen::Uniform(Uniform::new(5.0, 15.0))
                    ),
                    QueueResource::new(
                        PayloadQueue::default(),
                        3
```

```
                    )
                ))),
                Probability::new(0.5)
            );
            ProbabilityArray::<Node>::new(vec![node_prob.clone(),
node_prob])
        };
        let node = (
            Node::Process(RefCell::new(NodeProcess::new(
                NodeBase::new(
                    prob_array,
                    DelayGen::Uniform(Uniform::new(5.0, 15.0))
                ),
                QueueResource::new(
                    PayloadQueue::default(),
                    3
                )
            ))),
            Probability::new(0.1)
        );
        ProbabilityArray::<Node>::new(
            (0..10).map(|_| node.clone()).collect::<Vec<(Node,
Probability)>>()
        )
    };

    let tree = NodeCreate::new(
        NodeBase::new(prob_array, DelayGen::Uniform(Uniform::new(1.0, 3.0)))
    );

    let mut events = BinaryHeap::<Event>::new();
    events.push(Event::Create(tree.produce_event(TimePoint(0.0))));

    let end_time = TimePoint(100000.0);
    let _ = loop {
        let next_event = events.pop().unwrap();
        if next_event.get_current_t() > end_time {
            break next_event;
        }

        match next_event {
            Event::Create(event) => {
                let (event_self, next_event) = event.iterate();
                events.push(Event::Create(event_self));
```

```
                    if let Some(next_event) = next_event {
                        events.push(next_event)
                    }
                },
                Event::Process(event) => {
                    let (event_self, next_event) = event.iterate();
                    if let Some(event_self) = event_self {
                        events.push(Event::Process(event_self))
                    }
                    if let Some(next_event) = next_event {
                        events.push(next_event)
                    }
                }
            }
        };
    }
}

use std::cell::RefCell;
use std::collections::BinaryHeap;
use criterion::{criterion_group, criterion_main, BenchmarkId, Criterion};
use rand::distributions::Uniform;
use lab_4::{Event, Node, NodeBase, TimePoint};
use lab_4::delay_gen::DelayGen;
use lab_4::node_create::NodeCreate;
use lab_4::node_process::NodeProcess;
use lab_4::payload_queue::PayloadQueue;
use lab_4::prob_arr::{Probability, ProbabilityArray};
use lab_4::queue_resource::QueueResource;

fn simulate_model(mut events: BinaryHeap<Event>, end_time: TimePoint) {
    loop {
        let next_event = events.pop().unwrap();
        if next_event.get_current_t() > end_time {
            break next_event;
        }

        match next_event {
            Event::Create(event) => {
                let (event_self, next_event) = event.iterate();
                events.push(Event::Create(event_self));
                if let Some(next_event) = next_event {
                    events.push(next_event)
                }
```

```
            },
            Event::Process(event) => {
                let (event_self, next_event) = event.iterate();
                if let Some(event_self) = event_self {
                    events.push(Event::Process(event_self))
                }
                if let Some(next_event) = next_event {
                    events.push(next_event)
                }
            }
        }
    };
}

fn generate_recursive_sequential(depth: usize) -> (Node, Probability) {
    let prob_vec = if depth == 0 {
        ProbabilityArray::<Node>::new(
            vec![generate_recursive_sequential(depth - 1)]
        )
    } else {
        Default::default()
    };
    (
        Node::Process(RefCell::new(NodeProcess::new(
            NodeBase::new(prob_vec, DelayGen::Uniform(Uniform::new(5.0, 15.0))),
            QueueResource::new(PayloadQueue::default(), 10)
        ))),
        Probability::new(1.0)
    )
}

pub fn model_sequential(c: &mut Criterion) {
    let tree = NodeCreate::new(
        NodeBase::new(
            ProbabilityArray::<Node>::new(vec!
[generate_recursive_sequential(50)]),
            DelayGen::Uniform(Uniform::new(1.0, 3.0))
        )
    );

    let mut events = BinaryHeap::<Event>::new();
    events.push(Event::Create(tree.produce_event(TimePoint(0.0))));

    let mut group = c.benchmark_group("model_sequential");
```

```
    for size in (10000..=100000).step_by(10000) {
        group.bench_with_input(BenchmarkId::from_parameter(size), &size, |b, i|
b.iter(|| simulate_model(events.clone(), TimePoint(*i as f64))) );
    }
    group.finish();

}

pub fn model_trees_depth_2(c: &mut Criterion) {
    let prob_array = {
        let prob_array = {
            let node_prob = (
                Node::Process(RefCell::new(NodeProcess::new(
                    NodeBase::new(
                        Default::default(),
                        DelayGen::Uniform(Uniform::new(10.0, 15.0))
                    ),
                    QueueResource::new(
                        PayloadQueue::default(),
                        3
                    )
                ))),
                Probability::new(0.5)
            );
            ProbabilityArray::<Node>::new(vec![node_prob.clone(), node_prob])
        };
        let node = (
            Node::Process(RefCell::new(NodeProcess::new(
                NodeBase::new(
                    prob_array,
                    DelayGen::Uniform(Uniform::new(10.0, 15.0))
                ),
                QueueResource::new(
                    PayloadQueue::default(),
                    3
                )
            ))),
            Probability::new(0.1)
        );
        ProbabilityArray::<Node>::new(
            (0..10).map(|_| node.clone()).collect::<Vec<(Node, Probability)>>()
        )
    };
```

```
    let tree = NodeCreate::new(
        NodeBase::new(prob_array, DelayGen::Uniform(Uniform::new(1.0, 3.0)))
    );

    let mut events = BinaryHeap::<Event>::new();
    events.push(Event::Create(tree.produce_event(TimePoint(0.0))));

    let mut group = c.benchmark_group("model_trees_depth_2");
    for size in (10000..=100000).step_by(10000) {
        group.bench_with_input(BenchmarkId::from_parameter(size), &size, |b, i|
b.iter(|| simulate_model(events.clone(), TimePoint(*i as f64))) );
    }
    group.finish();
}

pub fn model_trees_depth_2_sequential(c: &mut Criterion) {
    let prob_array = {
        let prob_array = {
            let node_prob = (
                Node::Process(RefCell::new(NodeProcess::new(
                    NodeBase::new(
                        ProbabilityArray::<Node>::new(
                            vec![generate_recursive_sequential(50)]
                        ),
                        DelayGen::Uniform(Uniform::new(10.0, 15.0))
                    ),
                    QueueResource::new(
                        PayloadQueue::default(),
                        3
                    )
                ))),
                Probability::new(0.5)
            );
            ProbabilityArray::<Node>::new(vec![node_prob.clone(), node_prob])
        };
        let node = (
            Node::Process(RefCell::new(NodeProcess::new(
                NodeBase::new(
                    prob_array,
                    DelayGen::Uniform(Uniform::new(10.0, 15.0))
                ),
                QueueResource::new(
                    PayloadQueue::default(),
                    3
```

```
                )
            ))),
            Probability::new(0.1)
        );
        ProbabilityArray::<Node>::new(
            (0..10).map(|_| node.clone()).collect::<Vec<(Node, Probability)>>()
        )
    };

    let tree = NodeCreate::new(
        NodeBase::new(prob_array, DelayGen::Uniform(Uniform::new(1.0, 3.0)))
    );

    let mut events = BinaryHeap::<Event>::new();
    events.push(Event::Create(tree.produce_event(TimePoint(0.0))));

    let mut group = c.benchmark_group("model_trees_depth_2_sequential");
    for size in (10000..=100000).step_by(10000) {
        group.bench_with_input(BenchmarkId::from_parameter(size), &size, |b, i|
b.iter(|| simulate_model(events.clone(), TimePoint(*i as f64))) );
    }
    group.finish();
}

criterion_group!(benches, model_sequential, model_trees_depth_2,
model_trees_depth_2_sequential);
criterion_main!(benches);
```