У теорії пояснюються основи механізмів планування процесів у операційних системах, від низькорівневих механізмів (наприклад, перемикання контексту) до високорівневих політик, які ОС використовує для організації роботи процесів.

**Планувальні політики** – це підходи або стратегії, які операційна система застосовує для управління виконанням процесів. Планування існувало ще до комп'ютерних систем, запозичене з операційного менеджменту (наприклад, з виробничих ліній), де ефективність завжди була ключовою метою.

**Припущення про "навантаження" – процеси, які виконує система:**

* Усі завдання виконуються однаковий час.
* Усі завдання починають виконання одночасно.
* Завдання виконуються без перерви до завершення.
* Завдання використовують лише процесор (без вводу/виводу).
* Час виконання завдання заздалегідь відомий.

Ці припущення спрощують аналіз, але в реальному світі вони рідко правдиві.

Для оцінки ефективності планування введено **метрику часу обробки: Tturnaround = Tcompletion − Tarrival**​. На цьому етапі передбачено, що всі завдання надходять одночасно, тому Tarrival = 0, і метрика зводиться до часу завершення Tcompletion​. Іншою важливою метрикою є **справедливість**, але вона не є основною у цьому розділі.

### **Планувальні алгоритми:**

* **First In, First Out (FIFO):** процеси виконуються в порядку їх прибуття. Найпростіший і легкий у реалізації алгоритм. Проблема FIFO полягає у тому, що якщо одне завдання значно довше за інші, це призводить до "ефекту конвою" – короткі завдання чекають завершення довгих, навіть якщо вони прибули раніше.
* **Shortest Job First (SJF):** завдання з найкоротшим часом виконання запускається першим. Перевага полягає в тому, що значно зменшується середній час виконання завдань.

У попередніх системах обчислень використовувалися **невитіснювальні алгоритми** планування, де кожне завдання виконувалось до завершення. Сучасні ж системи використовують **витіснювальне планування**, що дозволяє зупинити одне завдання (контекстний перемикач) для запуску іншого, забезпечуючи більшу гнучкість. Алгоритм SJF оптимальний для середнього часу виконання, якщо всі завдання надходять одночасно. Проте, якщо завдання надходять із затримкою, виникає згаданий вище "ефект конвою". Щоб вирішити проблему, буде доцільно використати **Shortest Time-to-Completion First (STCF)** – модифікацію SJF, яка дозволяє перервати довше завдання і переключитися на коротші. Тобто при появі нового завдання планувальник вибирає те, яке має найменший залишковий час виконання.

### З появою інтерактивних систем з'явилася потреба в швидкій реакції на користувацькі запити. Метрика **час відповіді** визначається як час від прибуття завдання до його першого запуску: **Response = Tfirst run − Tarrival**.​ Якщо кілька завдань з'являються одночасно, останнє з них довго очікує першого запуску, що погано для інтерфейсів, тому STCF і SJF не підходять для цієї метрики. В цьому випадку буде доцільно використати **кругове планування (Round-Robin, RR)**, яке планує завдання на короткі часові квантові проміжки. Після закінчення кванта виконується контекстний перемикач, і CPU переходить до наступного завдання. Це забезпечує справедливість між завданнями та покращує час відповіді, але за рахунок збільшення часу виконання – занадто короткий квант призводить до великих витрат на перемикання контексту.

Отже, вибір планувальника залежить від пріоритету метрик (продуктивність проти інтерактивності).

### Алгоритм повинен враховувати **операції вводу/виводу (I/O)**, оскільки під час таких операцій CPU простоює. Рішенням у цій ситуації буде розбиття процесів на підзадачі. Планувальник обирає коротші завдання (STCF) для ефективності. Це дозволяє виконувати обчислення, поки завдання чекає завершення I/O і підвищувати загальну ефективність системи.

### Алгоритми SJF/STCF передбачають, що **тривалість** кожного завдання відома заздалегідь. У реальних ОС ця інформація недоступна. Тому виникає потреба в адаптивних алгоритмах, які динамічно оцінюють тривалість і комбінують найкращі аспекти SJF і RR.

### Згадуючи всі попередні тезіси, можна зробити висновок, що жоден підхід не є ідеальним: оптимізація одного критерію (часу завершення чи часу відповіді) може погіршувати інший. Фундаментальною проблемою є неможливість передбачити майбутнє.

### **Відповіді на запитання:**

1. **SJF:** Усі задачі мають однакову довжину, тому порядок їхнього виконання не змінюється: Job1 –> Job2 –> Job3. Turnaround time: [200, 400, 600]. Response time = [0, 200, 400]. **FIFO:** Порядок виконання ідентичний SJF.
2. **SJF:** Виконуються у порядку зростання довжини: Job1 (100) –> Job2 (200) –> Job3 (300). Turnaround time = [100, 300, 600]. Response time = [0, 100, 300]. **FIFO:** Виконуються у заданому порядку: Job1 (100) –> Job2 (200) –> Job3 (300). Turnaround time = [100, 300, 600]. Response time = [0, 100, 300].
3. Квант часу = 1 означає, що задачі чергуються, виконуючи по 1 одиниці часу. **Для задач довжиною 200:** Turnaround time = [400, 500, 600]. Response time = [0, 1, 2]. **Для задач довжиною 100, 200, 300:** Turnaround time = [299, 498, 600]. Response time = [0, 1, 2].
4. Коли всі завдання мають однакову довжину або вже подані в порядку зростання довжини, порядок виконання в SJF і FIFO ідентичний, тому часи завершення збігаються.
5. Якщо довжина кванта в RR наближається до довжини найкоротшого завдання в SJF, то часи відповіді можуть збігатися. Це можливо, коли завдання мають однакову довжину, і тайм-слот RR точно дорівнює цій довжині.
6. Час відповіді першої задачі залишається нульовим. Однак для наступних задач час відповіді значно зростає, оскільки вони виконуються лише після завершення коротших задач. **Демонстрація для задач довжиною 100, 200, 300, 400:** Response time = [0, 100, 300, 600].
7. При збільшенні кванту часу час відповіді першої задачі залишається нульовим, але час відповіді інших задач зростає, оскільки більші задачі "захоплюють" квант часу. **Формула для гіршого випадку: Response TimeWorst Case = (N − 1) ⋅ Quantum Length**, де N – кількість завдань.