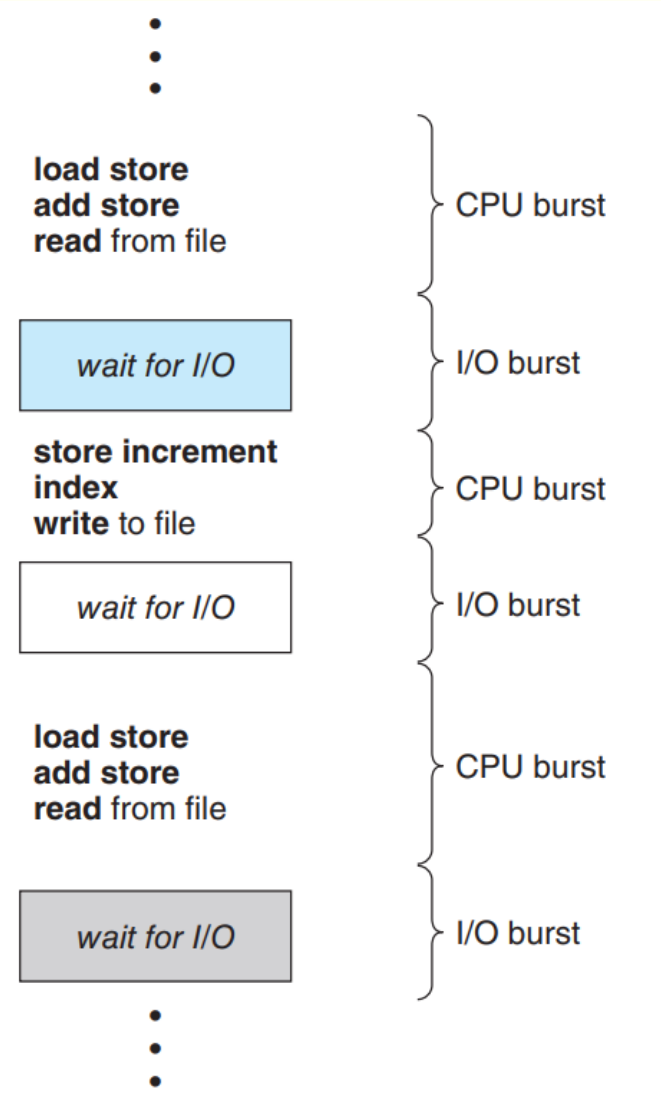
ПРИНЦИПИ РОБОТИ ПЛАНУВАЛЬНИКА ЦЕНТРАЛЬНОГО ПРОЦЕСОРА

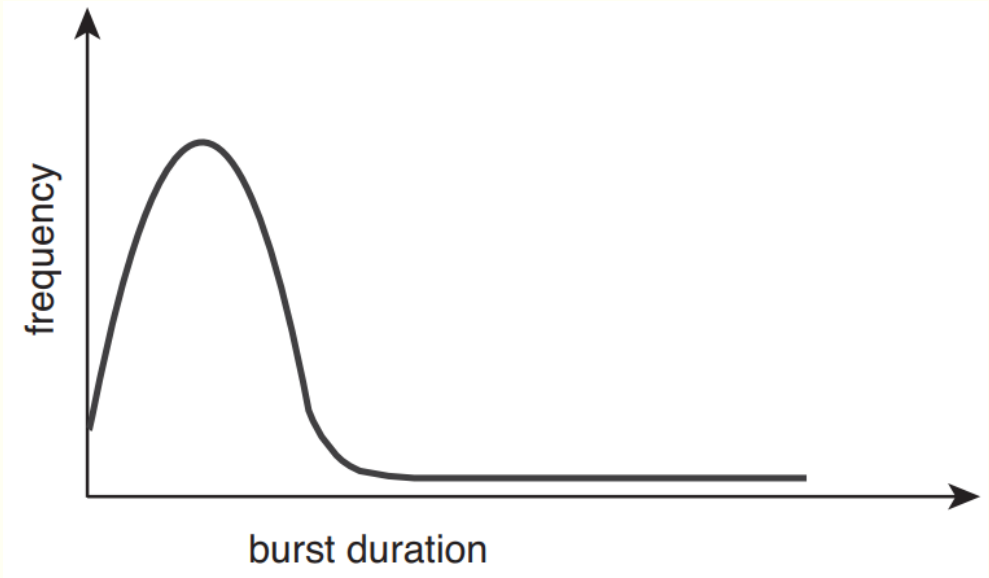
Питання 2.3

Базові поняття в плануванні процесів (CPU scheduling)



- На сучасних ОС фактично планується виконання kernel-level потоків, а не процесів.
  - Проте терміни "process scheduling" та "thread scheduling" часто взаємозамінні.
- Планування такого виду – фундаментальна функція ОС.
  - Майже для всіх ресурсів комп'ютера до їх використання відбувається планування.
- Успіх CPU scheduling залежить від спостережуваної властивості процесів: виконання процесу містить цикли виконання та очікування вводу-виводу.
  - Процеси перемикаються між цими двома станами.
  - Виконання процесу починається з неперервного використання ЦП ***(CPU burst)***, за ним іде неперервний ввід-вивід (I/O burst) і т. д.
  - Крайній CPU burst завершується системним запитом на переривання виконання.

Гістограма тривалості етапів CPU-burst



- Хоч графіки різноманітні залежно від процесу, частотна крива прямує до такої форми:
  - експоненційної або гіперекпоненційної, з великою кількістю коротких CPU bursts та малою кількістю тривалих CPU bursts.
  - Обмежена вводом-виводом (I/O-bound) програма зазвичай має багато коротких CPU bursts.
  - Обмежена ЦП (CPU-bound) програма може мати кілька тривалих CPU bursts. Такий розподіл може бути важливим при реалізації алгоритмів планування процесів.

- Як тільки ЦП починає простоювати, ОС повинна обрати 1 з процесів, готовий до виконання.
  - Процес вибору здійснюється планувальником ЦП **(CPU scheduler).**

- Зауважте, що черга готовності (ready queue) не обов'язково є FIFO-чергою.
  - Для різних алгоритмів планування ця черга може бути FIFO-чергою, чергою з пріоритетами, деревом або просто невпорядкованим зв'язним списком.
  - Записи в чергах загалом є процесними блоками управління (process control blocks, PCBs) процесу.

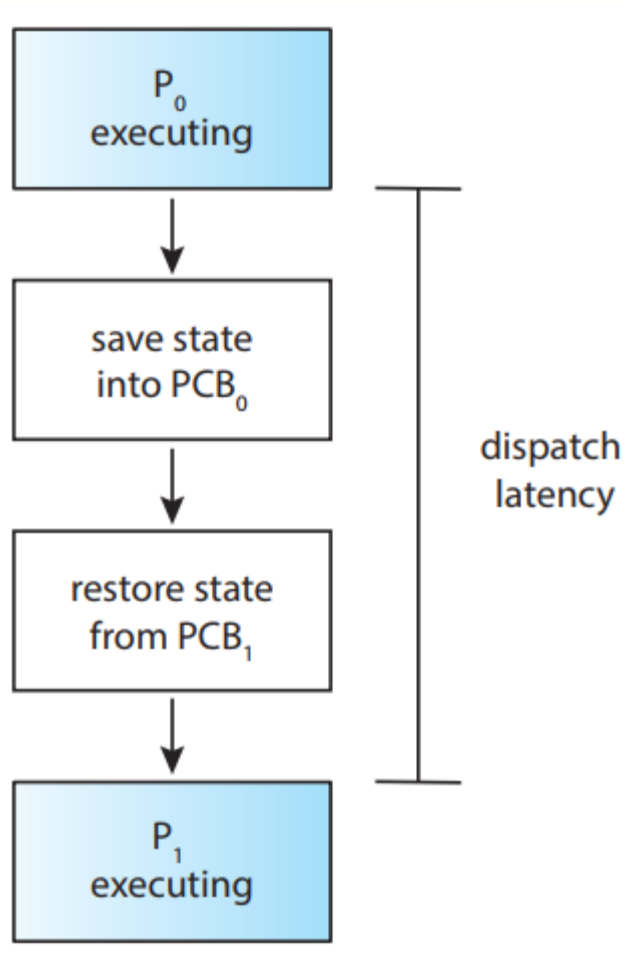
Витісняюче (Preemptive) та невитісняюче (Nonpreemptive) планування

- Рішення при плануванні процесів можуть здійснюватись в 4 випадках:
  - 1. Коли процес перемикається із запущеного стану в очікуючий (у результаті запиту на ввід-вивід, виклику wait() для переривання дочірнього процесу та ін.)
  - 2. Коли процес перемикається із запущеного стану в стан готовності (ready state) (наприклад, коли трапляється переривання)
  - 3. Коли процес перемикається зі стану очікування в стан готовності (наприклад, завершення вводу-виводу)
  - 4. Коли процес завершується (terminates)
- Для ситуацій 1 та 4 у контексті планування вибору немає.
  - Новий процес (якщо є в ready-черзі) повинен обиратись для виконання.
  - Коли планування відбувається тільки у випадках 1 і 4, схему планування називають ***невитісняючою (nonpreemptive, cooperative)*** – як тільки ЦП виділяється процесу, він утримує процесор, поки не трапиться переривання (terminating) або перехід у стан очікування.
  - Інакше схема витісняюча. Практично всі сучасні ОС використовують її.

На жаль, витісняюче планування може призводити до гонитви доступу до спільних даних для кількох процесів

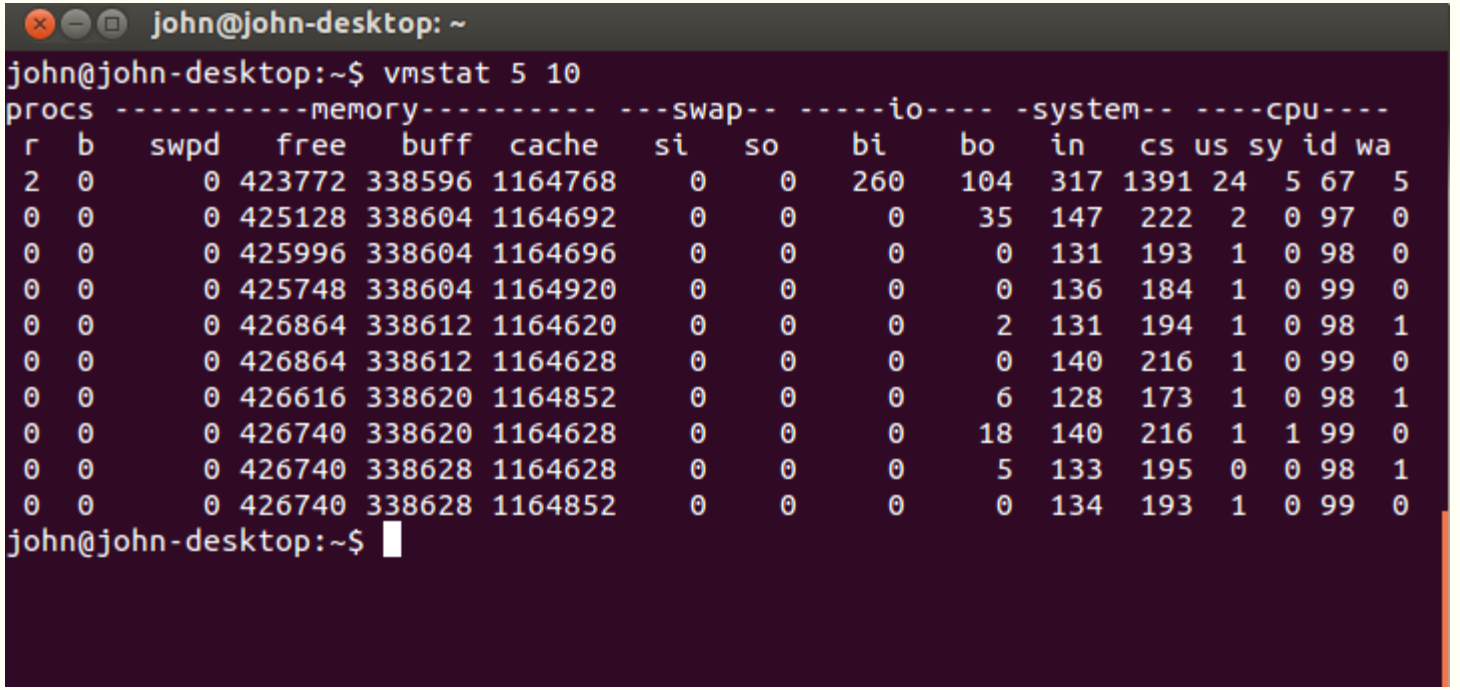
- Нехай є 2 процеси зі спільними даними.
  - Поки один процес оновлює дані, він витісняється іншим процесом, який намагається зчитати ці ж дані.
- Витіснення також впливає на проектування ядра ОС.
  - Протягом обробки системного виклику ядро може бути зайнятим with an activity on behalf of a process.
  - Такі дії (activities) можуть включати внесення змін у важливі дані ядра (наприклад, черги вводу-виводу).
  - Що трапляється, коли процес витісняється протягом цих змін, а ядру (чи драйверу пристрою) потрібно зчитати чи змінити ту ж структуру? *Настає хаос*.
- Невитісняюче ядро очікуватиме на завершення системного виклику або блокування процесу в очікуванні завершення вводу-виводу перед перемиканням контексту.
  - Схема забезпечує просту структуру ядра, оскільки воно не витіснятиме процес, поки структури даних ядра знаходяться в неузгодженому стані.
  - На жаль, ця модель слабка для підтримки обчислень у реальному часі, коли задачі повинні завершувати виконання протягом заданого часу.
  - Витісняюче ядро потребує механізму на зразок м'ютексів (mutex locks) для уникнення гонитви даних при доступі до спільних структур даних у ядрі.
  - Більшість сучасних ОС повністю витісняючі при роботі в режимі ядра.

Диспетчер



- Інший компонент, задіяний при плануванні процесів – ***диспетчер*** – модуль, який надає управління ядром ЦП обраному планувальником процесу.
- Дана функція включає:
  - Перемикання контексту від одного процесу до іншого
  - Перемикання в режим користувача
  - Перехід у коректне місце користувацької програми для відновлення її роботи
- Диспетчер має бути якомога швидшим, оскільки викликається при кожному перемиканні контексту.
  - Час, необхідний диспетчеру на зупинку одного процесу та запуск іншого, називають ***латентністю диспетчера (dispatch latency)***.
- Як часто відбуваються перемикання контексту?
  - На системному рівні кількість перемикань контексту можна отримати за допомогою команди vmstat на Linux-системах.

Команда vmstat (virtual memory statistics) – не працює на WLS



- 5 -> значення будуть перевимірюватись та оновлюватись кожні 5с
- 10 -> значення виведуться 10 разів, після чого програма зупинеться
  - Стовпчик cs – кількість перемикань контексту за секунду
  - Деталізована статистика: vmstat -s

Віртуальна файлова система /proc для обчислення кількості перемикань контексту заданого процесу

- Наприклад, вміст файлу /proc/2166/status виведе різну статистику для процесу з pid = 2166.
  - **`cat /proc/2166/status`** має наступний урізаний вивід:
    ```
    voluntary ctxt switches 150
    nonvoluntary ctxt switches 8
    ```
- Вивід показує кількість перемикань контексту протягом життя процесу.
  - ***Добровільне (voluntary) перемикання контексту*** відбувається тоді, коли процес передає управління центральним процесором, оскільки потребує на даний момент недоступних ресурсів, таких як blocking для вводу-виводу.
  - ***Примусове (nonvoluntary) перемикання контексту*** трапляється, коли ЦП забрали у процесу. Наприклад, коли закінчився виділений проміжок часу або його витіснив більш пріоритетний процес.

Критерії планування

- Різні алгоритми планування мають різні властивості, які потрібно враховувати при їх виборі.
  - **Задіяність ЦП.** Бажаємо завантажити ЦП по максимуму. Ідейно задіювання ЦП знаходиться в межах 0-100%, проте на практиці діапазон від 40% (легко завантажена система) до 90% (heavily loaded system). Отримати задіяність можна за допомогою команди `top` на Linux, macOS та UNIX.
  - **Пропускна здатність**. Однією з мір роботи, яку здійснює ЦП, є кількість процесів, завершених за період часу, який називають пропускною здатністю. Для тривалих процесів вона може бути 1 процес за кілька секунд, а для коротких транзакцій – десятки процесів/с.
  - **Turnaround time**. From the point of view of a particular process, the important criterion is how long it takes to execute that process. The interval from the time of submission of a process to the time of completion is the turnaround time. Turnaround time is the sum of the periods spent waiting in the ready queue, executing on the CPU, and doing I/O.
  - **Waiting time**. The CPU-scheduling algorithm does not affect the amount of time during which a process executes or does I/O. It affects only the amount of time that a process spends waiting in the ready queue. Waiting time is the sum of the periods spent waiting in the ready queue.
  - **Response time**. In an interactive system, turnaround time may not be the best criterion. Often, a process can produce some output fairly early and can continue computing new results while previous results are being output to the user. Thus, another measure is the time from the submission of a request until the first response is produced. This measure, called response time, is the time it takes to start responding, not the time it takes to output the response.

Критерії планування

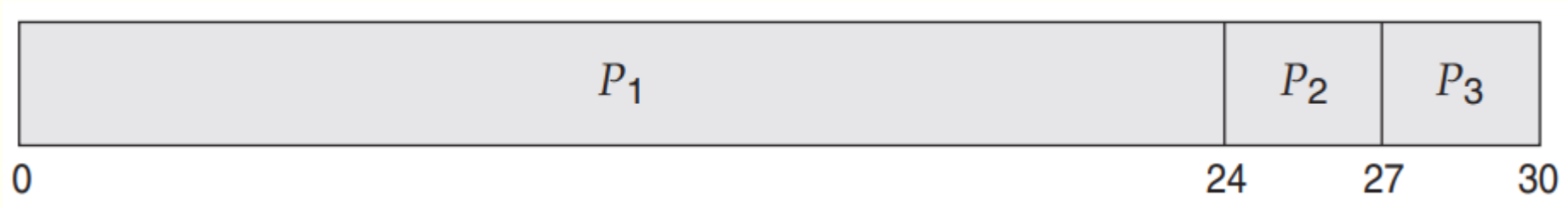
- It is desirable to maximize CPU utilization and throughput and to minimize turnaround time, waiting time, and response time.
  - In most cases, we optimize the average measure.
  - However, under some circumstances, we prefer to optimize the minimum or maximum values rather than the average.
  - For example, to guarantee that all users get good service, we may want to minimize the maximum response time.
- Investigators have suggested that, for interactive systems (such as a PC desktop or laptop system), it is more important to minimize the variance in the response time than to minimize the average response time.
  - A system with reasonable and predictable response time may be considered more desirable than a system that is faster on the average but is highly variable.
  - However, little work has been done on CPU-scheduling algorithms that minimize variance.
- As we discuss various CPU-scheduling algorithms in the following section, we illustrate their operation.
  - An accurate illustration should involve many processes, each a sequence of several hundred CPU bursts and I/O bursts.
  - For simplicity, though, we consider only one CPU burst (in milliseconds) per process in our examples.
  - Our measure of comparison is the average waiting time.

Алгоритми планування. First-Come, First-Served Scheduling

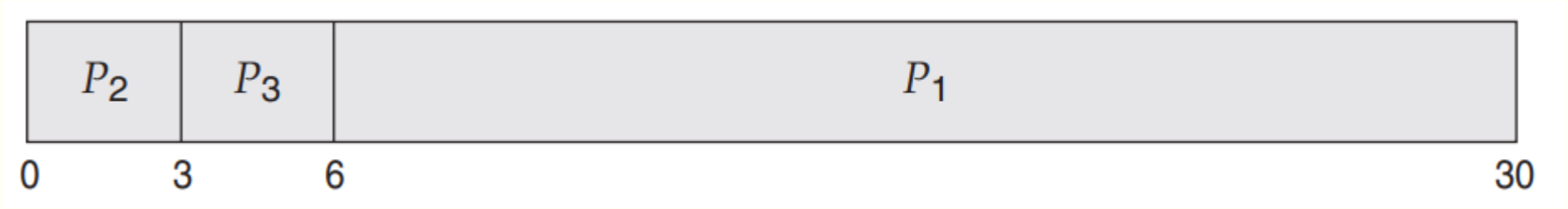
- Найпростіший алгоритм планування: процес, який першим подав запит на використання ЦП, першим його й отримує.
  - Політика FCFS легко реалізується за допомогою FIFO-черги.
  - Коли процес надходить до ready-черги, його PCB прикріпляється в хвіст черги.
  - Потім запущений процес видаляється з черги.
  - Код алгоритму простий для написання та розуміння.
  - Проте середній час очікування при FCFS-політиці часто досить тривалий.
- Розглянемо приклад з набором процесів, які надходять у момент часу $time = 0$ та мають відповідні тривалості неперервного використання ЦП (Burst time):

Process	Burst Time
P_1	24
P_2	3
P_3	3

Алгоритми планування. First-Come, First-Served Scheduling



- Час очікування складає 0мс для процесу P_1, 24мс для процесу P_2 та 27мс для процесу P_3.
  - Середній час очікування: (0 + 24 + 27)/3 = 17мс.
- Якщо процеси надходять у порядку P_2, P_3, P_1:



  - Середній час очікування: (6 + 0 + 3)/3 = 3мс.
  - Середній час очікування при FCFS-політиці планування загалом не мінімальний та суттєво залежить від того, наскільки відрізняються тривалості CPU burst для процесів.

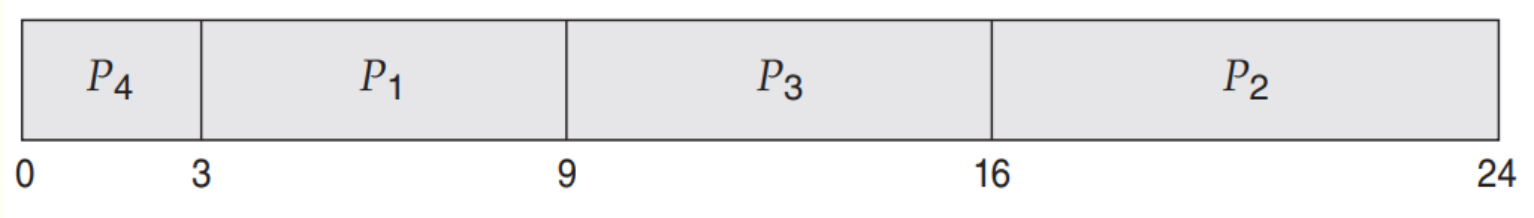
Алгоритми планування. First-Come, First-Served Scheduling

- Розглянемо продуктивність FCFS-планування в динамічній ситуації для одного CPU-bound процесу та багатьох I/O-bound процесів.
  - Може настати такий сценарій: CPU-bound процес утримуватиме ЦП, а інші процеси завершать свої операції вводу-виводу та перемістяться в ready-чергу, очікуючи доступність ЦП. У цей час пристрої вводу-виводу не задіяні (idle).
  - З часом CPU-bound процес завершить свій CPU burst та перейде до I/O device. Усі I/O-bound процеси (з короткими CPU bursts) виконуються швидко та повертаються назад у черги вводу-виводу.
  - У цей момент ЦП простоює. CPU-bound процес will then move back to the ready queue and be allocated the CPU.
- Ще раз, усі процеси вводу-виводу закінчують очікуванням у ready-черзі, поки CPU-bound процес не буде виконано.
  - There is a **convoy effect** as all the other processes wait for the one big process to get off the CPU.
  - This effect results in lower CPU and device utilization than might be possible if the shorter processes were allowed to go first.
- Note also that the FCFS scheduling algorithm is nonpreemptive.
  - Once the CPU has been allocated to a process, that process keeps the CPU until it releases the CPU, either by terminating or by requesting I/O.
  - The FCFS algorithm is thus particularly troublesome for interactive systems, where it is important that each process get a share of the CPU at regular intervals.
  - It would be disastrous to allow one process to keep the CPU for an extended period.

Алгоритми планування. Shortest-Job-First Scheduling

- This algorithm associates with each process the length of the process's next CPU burst.
  - When the CPU is available, it is assigned to the process that has the smallest next CPU burst.
  - If the next CPU bursts of two processes are the same, FCFS scheduling is used to break the tie.
  - Note that a more appropriate term for this scheduling method would be the ***shortest-next-CPU-burst*** algorithm, because scheduling depends on the length of the next CPU burst of a process, rather than its total length.
  - We use the term SJF because most people and textbooks use this term to refer to this type of scheduling.
- As an example of SJF scheduling, consider the following set of processes, with the length of the CPU burst given in milliseconds:
  - The waiting time is 3 milliseconds for process *P*1, 16 milliseconds for process *P*2, 9 milliseconds for process *P*3, and 0 milliseconds for process *P*4.
  - Thus, the average waiting time is (3 + 16 + 9 + 0)/4 = 7 milliseconds.
  - By comparison, if we were using the FCFS scheduling scheme, the average waiting time would be 10.25мс.

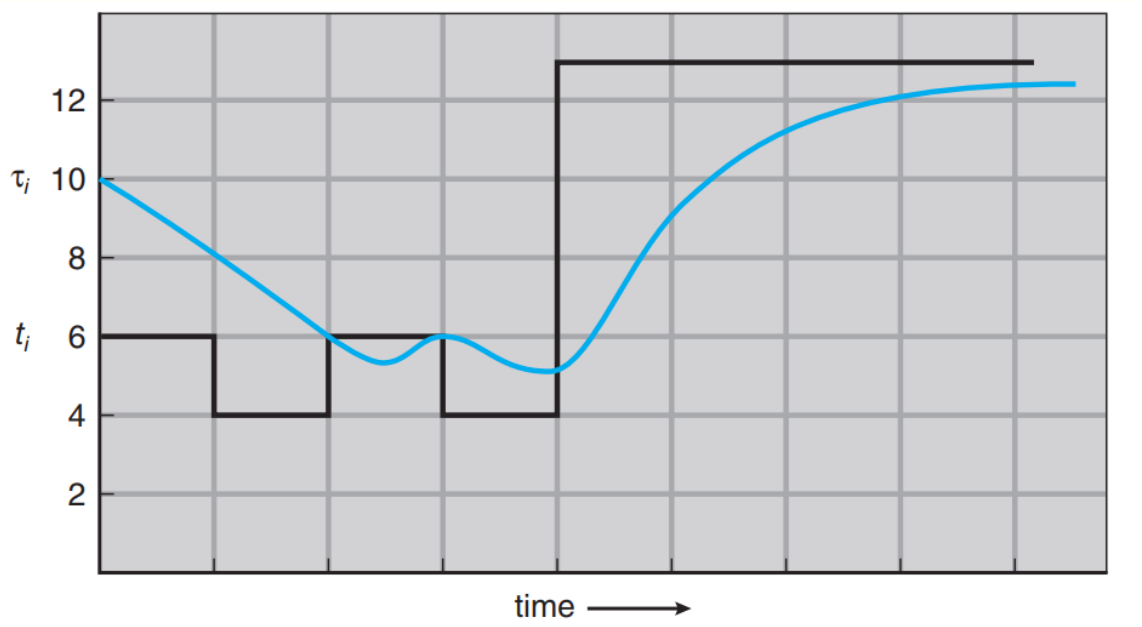
Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3



Алгоритми планування. Shortest-Job-First Scheduling

- The SJF scheduling algorithm is provably optimal, in that it gives the minimum average waiting time for a given set of processes.
  - Moving a short process before a long one decreases the waiting time of the short process more than it increases the waiting time of the long process.
  - Consequently, the average waiting time decreases.
- Although the SJF algorithm is optimal, it cannot be implemented at the level of CPU scheduling, as there is no way to know the length of the next CPU burst.
  - One approach to this problem is to try to approximate SJF scheduling.
  - We may not know the length of the next CPU burst, but we may be able to predict its value.
  - We expect that the next CPU burst will be similar in length to the previous ones.
  - By computing an approximation of the length of the next CPU burst, we can pick the process with the shortest predicted CPU burst.

Алгоритми планування. Shortest-Job-First Scheduling



CPU burst (t_i)	6	4	6	4	13	13	13	...
"guess" (τ_i)	10	8	6	6	5	9	11	12	...

The next CPU burst is generally predicted as an **exponential average** of the measured lengths of previous CPU bursts.

- We can define the exponential average with the following formula.
- Let t_n be the length of the *n*th CPU burst, and let τ_{n+1} be our predicted value for the next CPU burst.
- Then, for α, $0 \leq \alpha \leq 1$, define

$$\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$$

- The value of t_n contains our most recent information, while τ_n stores the past history.
- The parameter α controls the relative weight of recent and past history in our prediction.
- If $\alpha = 0$, then $\tau_{n+1} = \tau_n$, and recent history has no effect (current conditions are assumed to be transient).
- If $\alpha = 1$, then $\tau_{n+1} = t_n$, and only the most recent CPU burst matters (history is assumed to be old and irrelevant).
- More commonly, $\alpha = 1/2$, so recent history and past history are equally weighted.
- The initial τ_0 can be defined as a constant or as an overall system average.

Операції над процесами. Створення процесу

- To understand the behavior of the exponential average, we can expand the formula for τ_{n+1} by substituting for τ_n to find:

$$\tau_{n+1} = \alpha t_n + (1 - \alpha)\alpha t_n - 1 + \cdots + (1 - \alpha)^j \alpha t_{n-j} + \cdots + (1 - \alpha)^{n+1} \tau_0$$

  - Typically, α is less than 1. As a result, (1 – α) is also less than 1, and each successive term has less weight than its predecessor.

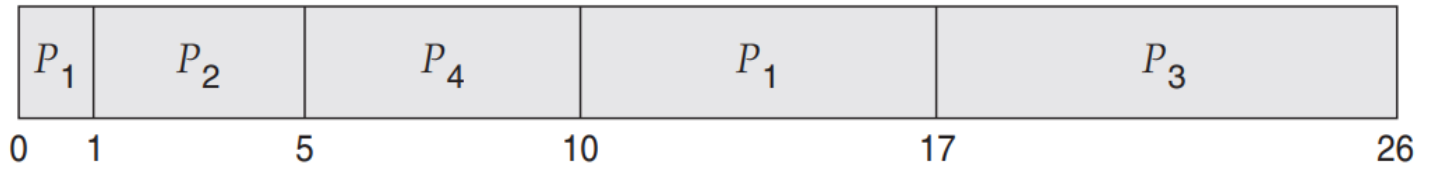
- The SJF algorithm can be either preemptive or nonpreemptive.
  - The choice arises when a new process arrives at the ready queue while a previous process is still executing.
  - The next CPU burst of the newly arrived process may be shorter than what is left of the currently executing process.
  - A preemptive SJF algorithm will preempt the currently executing process, whereas a nonpreemptive SJF algorithm will allow the currently running process to finish its CPU burst.
  - Preemptive SJF scheduling is sometimes called ***shortest-remaining time-first scheduling***.

Приклад

- consider the following four processes, with the length of the CPU burst given in milliseconds:

Process	Arrival Time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

- If the processes arrive at the ready queue at the times shown and need the indicated burst times, then the resulting preemptive SJF schedule



  - Process P1 is started at time 0, since it is the only process in the queue.
  - Process P2 arrives at time 1.
  - The remaining time for process P1 (7 milliseconds) is larger than the time required by process P2 (4 milliseconds), so process P1 is preempted, and process P2 is scheduled.
  - The average waiting time for this example is [(10 – 1) + (1 – 1) + (17 – 2) + (5 – 3)]/4 = 26/4 = 6.5 milliseconds.
  - Nonpreemptive SJF scheduling would result in an average waiting time of 7.75мс.

Алгоритм циклічної диспетчеризації (Round-Robin Scheduling)

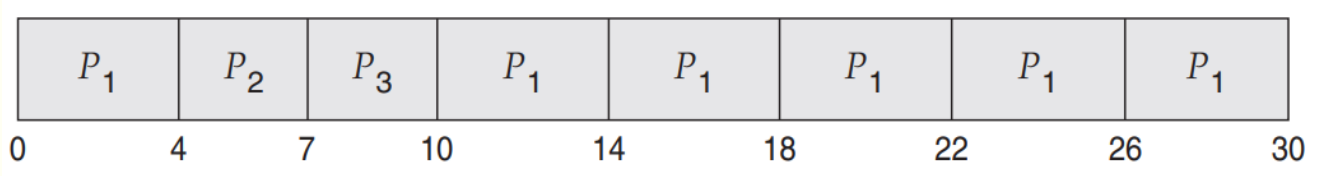
- The **round-robin** (**RR**) scheduling algorithm is similar to FCFS scheduling, but preemption is added to enable the system to switch between processes.
  - A small unit of time, called a **time quantum** or **time slice**, is defined.
  - A time quantum is generally from 10 to 100 milliseconds in length.
  - The ready queue is treated as a circular queue. The CPU scheduler goes around the ready queue, allocating the CPU to each process for a time interval of up to 1 time quantum.
- To implement RR scheduling, we again treat the ready queue as a FIFO queue of processes.
  - New processes are added to the tail of the ready queue.
  - The CPU scheduler picks the first process from the ready queue, sets a timer to interrupt after 1 time quantum, and dispatches the process.
- One of two things will then happen.
  - The process may have a CPU burst of less than 1 time quantum. In this case, the process itself will release the CPU voluntarily. The scheduler will then proceed to the next process in the ready queue. If the CPU burst of the currently running process is longer than 1 time quantum, the timer will go off and will cause an interrupt to the operating system. A context switch will be executed, and the process will be put at the tail of the ready queue. The CPU scheduler will then select the next process in the ready queue.

The average waiting time under the RR policy is often long.

- Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds:

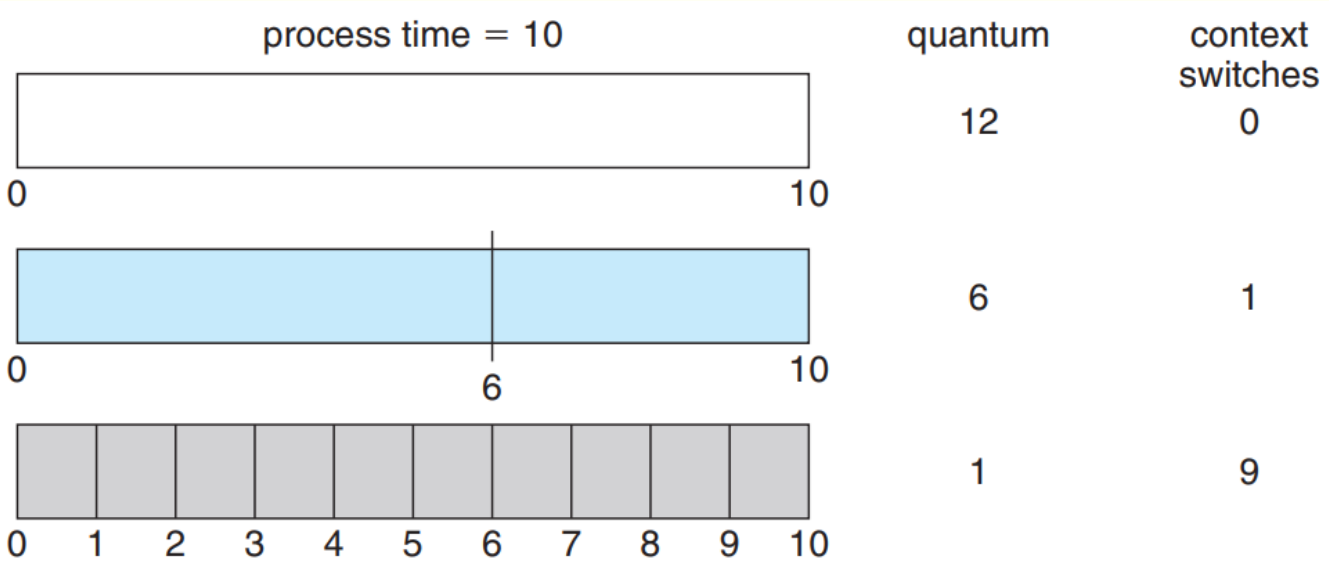
Process	Burst Time
P_1	24
P_2	3
P_3	3

- If we use a time quantum of 4 milliseconds, then process *P*1 gets the first 4 milliseconds.
  - Since it requires another 20 milliseconds, it is preempted after the first time quantum, and the CPU is given to the next process in the queue, process *P*2.
  - Process *P*2 does not need 4 milliseconds, so it quits before its time quantum expires.
  - The CPU is then given to the next process, process *P*3.
  - Once each process has received 1 time quantum, the CPU is returned to process *P*1 for an additional time quantum.
  - Let's calculate the average waiting time for this schedule. P1 waits for 6 milliseconds (10 − 4), P2 waits for 4 milliseconds, and P3 waits for 7 milliseconds.
  - Thus, the average waiting time is 17/3 = 5.66 milliseconds.



In the RR scheduling algorithm, no process is allocated the CPU for more than 1 time quantum in a row (unless it is the only runnable process).

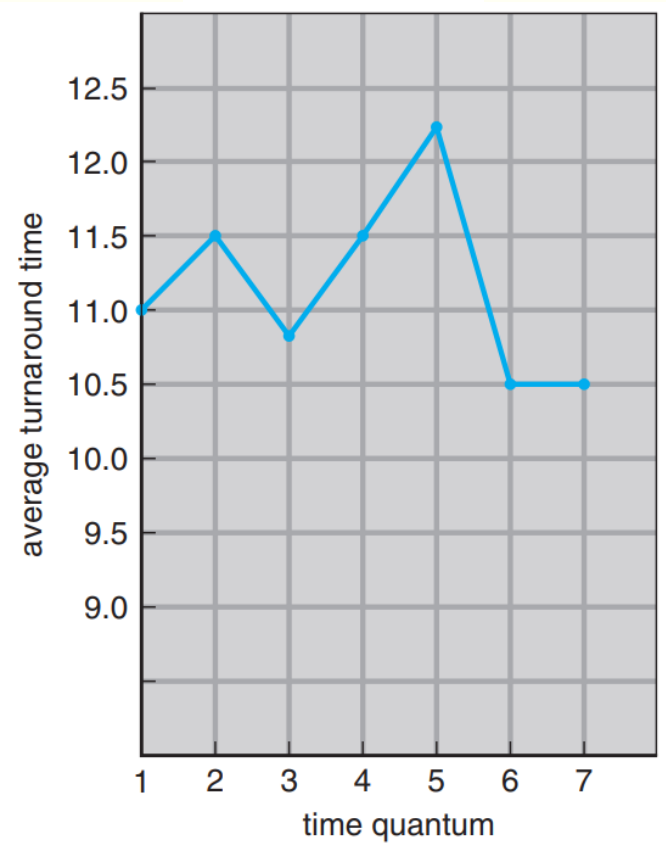
- If a process's CPU burst exceeds 1 time quantum, that process is preempted and is put back in the ready queue. The RR scheduling algorithm is thus preemptive.
- If there are *n* processes in the ready queue and the time quantum is *q*, then each process gets 1/*n* of the CPU time in chunks of at most *q* time units.
  - Each process must wait no longer than (*n* - 1) × *q* time units until its next time quantum.
  - For example, with five processes and a time quantum of 20 milliseconds, each process will get up to 20 milliseconds every 100 milliseconds.



- Продуктивність алгоритму RR дуже залежить від розміру кванту часу.
  - При дуже великому кванті політика вироджується в FCFS. При дуже малому (десь 1мс) RR-підхід призведе до великої кількості перемикань контексту.
- Нехай матимемо only one process of 10 time units.
  - If the time quantum is 1 time unit, then nine context switches will occur, slowing the execution of the process accordingly.

we want the time quantum to be large with respect to the context-switch time

process	time
P_1	6
P_2	3
P_3	1
P_4	7



- If the context-switch time is approximately 10 percent of the time quantum, then about 10% of the CPU time will be spent in context switching.
  - In practice, most modern systems have time quanta ranging from 10 to 100 milliseconds.
  - The time required for a context switch is typically less than 10 microseconds; thus, the context-switch time is a small fraction of the time quantum.
- Turnaround time also depends on the size of the time quantum.
  - the average turnaround time of a set of processes does not necessarily improve as the time-quantum size increases.
  - In general, the average turnaround time can be improved if most processes finish their next CPU burst in a single time quantum.
- For example, given three processes of 10 time units each and a quantum of 1 time unit, the average turnaround time is 29.
  - If the time quantum is 10, however, the average turnaround time drops to 20.
  - If context-switch time is added in, the average turnaround time increases even more for a smaller time quantum, since more context switches are required.
- Although the time quantum should be large compared with the context switch time, it should not be too large.
  - As we pointed out earlier, if the time quantum is too large, RR scheduling degenerates to an FCFS policy.
  - A rule of thumb is that 80% of the CPU bursts should be shorter than the time quantum.

Пріоритетне планування

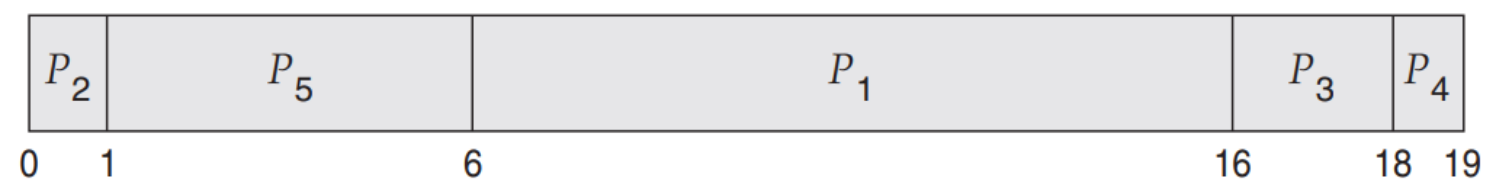
- The SJF algorithm is a special case of the general priority-scheduling algorithm.
  - A priority is associated with each process, and the CPU is allocated to the process with the highest priority. Equal-priority processes are scheduled in FCFS order.
  - An SJF algorithm is simply a priority algorithm where the priority (*p*) is the inverse of the (predicted) next CPU burst.
  - The larger the CPU burst, the lower the priority, and vice versa.
- Note that we discuss scheduling in terms of ***high*** priority and ***low*** priority.
  - Priorities are generally indicated by some fixed range of numbers, such as 0 to 7 or 0 to 4,095.
  - However, there is no general agreement on whether 0 is the highest or lowest priority.
  - Some systems use low numbers to represent low priority; others use low numbers for high priority.
  - This difference can lead to confusion. In this text, we assume that low numbers represent high priority.

Приклад

- consider the following set of processes, assumed to have arrived at time 0 in the order P1, P2, · · ·, P5, with the length of the CPU burst given in milliseconds:

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

  - Priorities can be defined either internally or externally. Internally defined priorities use some measurable quantity or quantities to compute the priority of a process.
  - For example, time limits, memory requirements, the number of open files, and the ratio of average I/O burst to average CPU burst have been used in computing priorities.
  - External priorities are set by criteria outside the operating system, such as the importance of the process, the type and amount of funds being paid for computer use, the department sponsoring the work, and other, often political, factors.



Priority scheduling can be either preemptive or nonpreemptive.

- When a process arrives at the ready queue, its priority is compared with the priority of the currently running process.
  - A preemptive priority scheduling algorithm will preempt the CPU if the priority of the newly arrived process is higher than the priority of the currently running process.
  - A nonpreemptive priority scheduling algorithm will simply put the new process at the head of the ready queue.
- A major problem with priority scheduling algorithms is **indefinit blocking**, or **starvation**.
- A process that is ready to run but waiting for the CPU can be considered blocked.
  - A priority scheduling algorithm can leave some lowpriority processes waiting indefinitely.
  - In a heavily loaded computer system, a steady stream of higher-priority processes can prevent a low-priority process from ever getting the CPU.
  - Generally, one of two things will happen.
  - Either the process will eventually be run (at 2 A.M. Sunday, when the system is finally lightly loaded), or the computer system will eventually crash and lose all unfinished low-priority processes.
  - (Rumor has it that when they shut down the IBM 7094 at MIT in 1973, they found a low-priority process that had been submitted in 1967 and had not yet been run.)

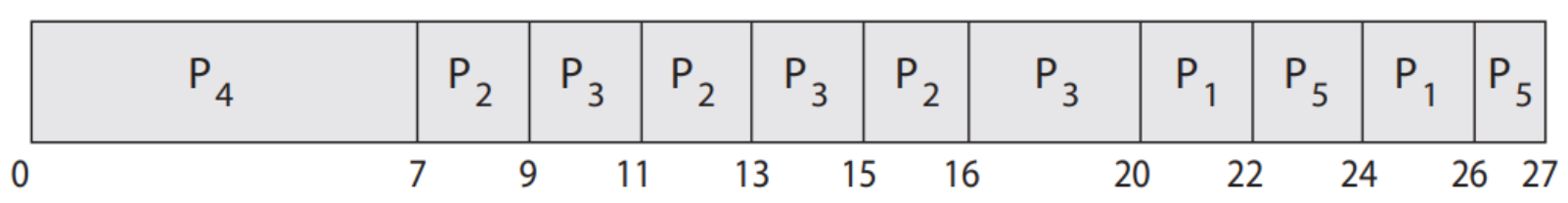
A solution to the problem of indefinite blockage of low-priority processes is aging.

- Aging involves gradually increasing the priority of processes that wait in the system for a long time.
  - For example, if priorities range from 127 (low) to 0 (high), we could periodically (say, every second) increase the priority of a waiting process by 1.
  - Eventually, even a process with an initial priority of 127 would have the highest priority in the system and would be executed.
  - In fact, it would take a little over 2 minutes for a priority-127 process to age to a priority-0 process.
- Another option is to combine round-robin and priority scheduling in such a way that the system executes the highest-priority process and runs processes with the same priority using round-robin scheduling.

Приклад

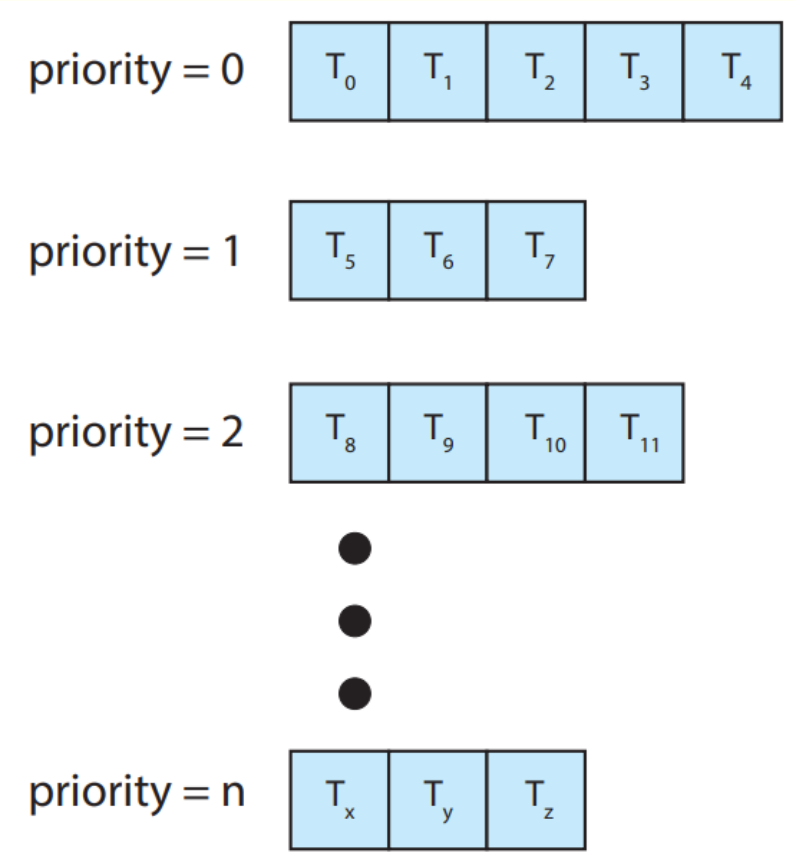
Process	Burst Time	Priority
P_1	4	3
P_2	5	2
P_3	8	2
P_4	7	1
P_5	3	3

- Using priority scheduling with round-robin for processes with equal priority, we would schedule these processes according to the following Gantt chart using a time quantum of 2 milliseconds:



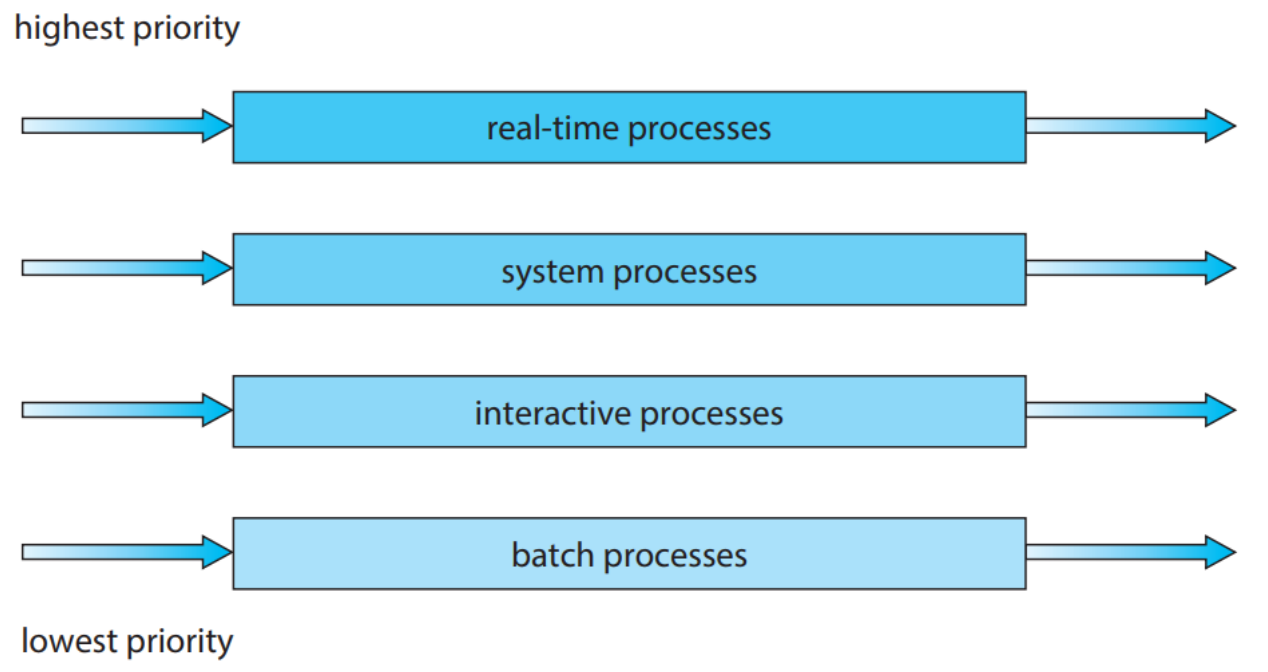
  - In this example, process *P*4 has the highest priority, so it will run to completion.
  - Processes *P*2 and *P*3 have the next-highest priority, and they will execute in a round-robin fashion.
  - Notice that when process *P*2 finishes at time 16, process *P*3 is the highest-priority process, so it will run until it completes execution.
  - Now, only processes *P*1 and *P*5 remain, and as they have equal priority, they will execute in round-robin order until they complete.

Багаторівневі черги (Multilevel Queue Scheduling)



- With both priority and round-robin scheduling, all processes may be placed in a single queue, and the scheduler then selects the process with the highest priority to run.
  - Depending on how the queues are managed, an *O*(*n*) search may be necessary to determine the highest-priority process.
  - In practice, it is often easier to have separate queues for each distinct priority, and priority scheduling simply schedules the process in the highest-priority queue.
  - This approach—known as **multilevel queue**—also works well when priority scheduling is combined with round-robin: if there are multiple processes in the highest-priority queue, they are executed in round-robin order.
  - In the most generalized form of this approach, a priority is assigned statically to each process, and a process remains in the same queue for the duration of its runtime.

Багаторівневі черги (Multilevel Queue Scheduling)



- In addition, there must be scheduling among the queues, which is commonly implemented as fixed-priority preemptive scheduling.
  - For example, the real-time queue may have absolute priority over the interactive queue.

- A multilevel queue scheduling algorithm can also be used to partition processes into several separate queues based on the process type.
  - For example, a common division is made between **foreground** (interactive) processes and **background** (batch) processes.
  - These two types of processes have different response-time requirements and so may have different scheduling needs.
  - In addition, foreground processes may have priority (externally defined) over background processes.
  - Separate queues might be used for foreground and background processes, and each queue might have its own scheduling algorithm.
  - The foreground queue might be scheduled by an RR algorithm, for example, while the background queue is scheduled by an FCFS algorithm.

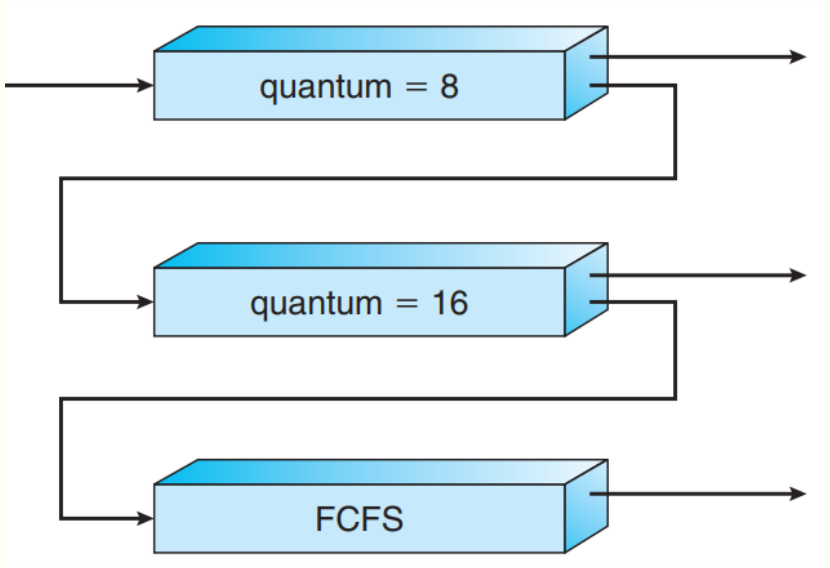
Багаторівневі черги (Multilevel Queue Scheduling)

- Розглянемо приклад multilevel queue scheduling algorithm з 4ма чергами:
  - **1.** Real-time processes
  - **2.** System processes
  - **3.** Interactive processes
  - **4.** Batch processes
- Each queue has absolute priority over lower-priority queues.
  - No process in the batch queue, for example, could run unless the queues for real-time processes, system processes, and interactive processes were all empty.
  - If an interactive process entered the ready queue while a batch process was running, the batch process would be preempted.
- Another possibility is to time-slice among the queues.
  - Here, each queue gets a certain portion of the CPU time, which it can then schedule among its various processes.
  - For instance, in the foreground–background queue example, the foreground queue can be given 80% of the CPU time for RR scheduling among its processes, while the background queue receives 20% of the CPU to give to its processes on an FCFS basis.

Multilevel Feedback Queue Scheduling

- Normally, when the multilevel queue scheduling algorithm is used, processes are permanently assigned to a queue when they enter the system.
  - If there are separate queues for foreground and background processes, for example, processes do not move from one queue to the other, since processes do not change their foreground or background nature.
  - This setup has the advantage of low scheduling overhead, but it is inflexible.
- The **multilevel feedback queue** scheduling algorithm, in contrast, allows a process to move between queues.
  - The idea is to separate processes according to the characteristics of their CPU bursts.
  - If a process uses too much CPU time, it will be moved to a lower-priority queue.
  - This scheme leaves I/O-bound and interactive processes—which are typically characterized by short CPU bursts—in the higher-priority queues.
  - In addition, a process that waits too long in a lower-priority queue may be moved to a higher-priority queue. This form of aging prevents starvation.

Multilevel Feedback Queue Scheduling



- consider a multilevel feedback queue scheduler with three queues, numbered from 0 to 2.
  - The scheduler first executes all processes in queue 0.
  - Only when queue 0 is empty will it execute processes in queue 1.
  - Similarly, processes in queue 2 will be executed only if queues 0 and 1 are empty.
  - A process that arrives for queue 1 will preempt a process in queue 2.
  - A process in queue 1 will in turn be preempted by a process arriving for queue 0.
- An entering process is put in queue 0.
  - A process in queue 0 is given a time quantum of 8 milliseconds.
  - If it does not finish within this time, it is moved to the tail of queue 1.
  - If queue 0 is empty, the process at the head of queue 1 is given a quantum of 16 milliseconds.
  - If it does not complete, it is preempted and is put into queue 2.
  - Processes in queue 2 are run on an FCFS basis but are run only when queues 0 and 1 are empty.
  - To prevent starvation, a process that waits too long in a lower-priority queue may gradually be moved to a higher-priority queue.

Multilevel Feedback Queue Scheduling

- This scheduling algorithm gives highest priority to any process with a CPU burst of 8 milliseconds or less.
  - Such a process will quickly get the CPU, finish its CPU burst, and go off to its next I/O burst.
  - Processes that need more than 8 but less than 24 milliseconds are also served quickly, although with lower priority than shorter processes.
  - Long processes automatically sink to queue 2 and are served in FCFS order with any CPU cycles left over from queues 0 and 1.
- In general, a multilevel feedback queue scheduler is defined by the following parameters:
  - The number of queues
  - The scheduling algorithm for each queue
  - The method used to determine when to upgrade a process to a higherpriority queue
  - The method used to determine when to demote a process to a lowerpriority queue
  - The method used to determine which queue a process will enter when that process needs service
- The definition of a multilevel feedback queue scheduler makes it the most general CPU-scheduling algorithm.
  - It can be configured to match a specific system under design.
  - Unfortunately, it is also the most complex algorithm, since defining the best scheduler requires some means by which to select values for all the parameters.

Планування потоків. Contention Scope

- One distinction between user-level and kernel-level threads lies in how they are scheduled.
  - On systems implementing the many-to-one and many-to-many models, the thread library schedules user-level threads to run on an available LWP.
  - This scheme is known as **process-contention scope** (**PCS**), since competition for the CPU takes place among threads belonging to the same process.
  - (When we say the thread library *schedules* user threads onto available LWPs, we do not mean that the threads are actually *running* on a CPU as that further requires the operating system to schedule the LWP's kernel thread onto a physical CPU core.)
- To decide which kernel-level thread to schedule onto a CPU, the kernel uses **system-contention scope** (**SCS**).
  - Competition for the CPU with SCS scheduling takes place among all threads in the system.
  - Systems using the one-to-one model, such as Windows and Linux schedule threads using only SCS.
- Typically, PCS is done according to priority—the scheduler selects the runnable thread with the highest priority to run.
  - User-level thread priorities are set by the programmer and are not adjusted by the thread library, although some thread libraries may allow the programmer to change the priority of a thread.
  - It is important to note that PCS will typically preempt the thread currently running in favor of a higher-priority thread; however, there is no guarantee of time slicing among threads of equal priority.

Pthread Scheduling

- we highlight the POSIX Pthread API that allows specifying PCS or SCS during thread creation.
  - Pthreads identifies the following contention scope values:
    - PTHREAD SCOPE PROCESS schedules threads using PCS scheduling.
    - PTHREAD SCOPE SYSTEM schedules threads using SCS scheduling.
- On systems implementing the many-to-many model, the PTHREAD_SCOPE_PROCESS policy schedules user-level threads onto available LWPs.
  - The number of LWPs is maintained by the thread library, perhaps using scheduler activations.
  - The PTHREAD_SCOPE_SYSTEM scheduling policy will create and bind an LWP for each user-level thread on many-to-many systems, effectively mapping threads using the one-to-one policy.
- The Pthread IPC (Interprocess Communication) provides two functions for setting—and getting—the contention scope policy:
  - • pthread attr setscope(pthread attr t *attr, int scope)
  - • pthread attr getscope(pthread attr t *attr, int *scope)
- The first parameter for both functions contains a pointer to the attribute set for the thread.
  - The second parameter for the pthread_attr_setscope() function is passed either the PTHREAD_SCOPE_SYSTEM or the PTHREAD_SCOPE_PROCESS value, indicating how the contention scope is to be set.
  - In the case of pthread attr getscope(), this second parameter contains a pointer to an int value that is set to the current value of the contention scope.
  - If an error occurs, each of these functions returns a nonzero value.

```c
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5

int main(int argc, char *argv[])
{
   int i, scope;
   pthread_t tid[NUM_THREADS];
   pthread_attr_t attr;

   /* get the default attributes */
   pthread_attr_init(&attr);

   /* first inquire on the current scope */
   if (pthread_attr_getscope(&attr, &scope) != 0)
      fprintf(stderr, "Unable to get scheduling scope\n");
   else {
      if (scope == PTHREAD_SCOPE_PROCESS)
       printf("PTHREAD_SCOPE_PROCESS");
      else if (scope == PTHREAD_SCOPE_SYSTEM)
       printf("PTHREAD_SCOPE_SYSTEM");
      else
       fprintf(stderr, "Illegal scope value.\n");
    }

   /* set the scheduling algorithm to PCS or SCS */
   pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);

   /* create the threads */
   for (i = 0; i < NUM_THREADS; i++)
       pthread_create(&tid[i],&attr,runner,NULL);

   /* now join on each thread */
   for (i = 0; i < NUM_THREADS; i++)
       pthread_join(tid[i], NULL);
}
```

illustrate a Pthread scheduling API

- The program first determines the existing contention scope and sets it to PTHREAD_SCOPE_SYSTEM.
  - It then creates five separate threads that will run using the SCS scheduling policy.
  - Note that on some systems, only certain contention scope values are allowed.
  - For example, Linux and macOS systems allow only PTHREAD_SCOPE_SYSTEM.

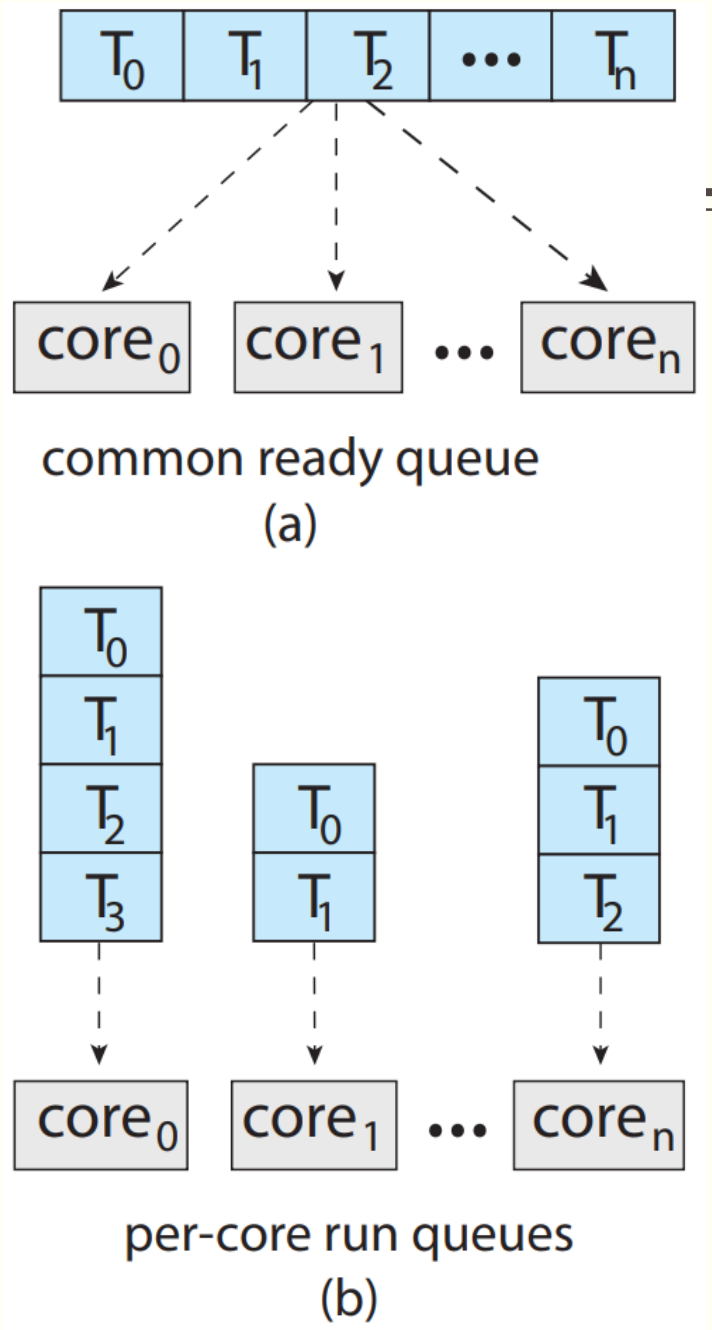
```c
/* Each thread will begin control in this function */
void *runner(void *param)
{
   /* do some work ... */

   pthread_exit(0);
}
```

Multi-Processor Scheduling

- If multiple CPUs are available, **load sharing**, where multiple threads may run in parallel, becomes possible, however scheduling issues become correspondingly more complex.
  - Many possibilities have been tried; and as we saw with CPU scheduling with a single-core CPU, there is no one best solution.
- Traditionally, the term **multiprocessor** referred to systems that provided multiple physical processors, where each processor contained one single-core CPU.
- However, the definition of multiprocessor has evolved significantly, and on modern computing systems, ***multiprocessor*** now applies to the following system architectures:
  • Multicore CPUs
  • Multithreaded cores
  • NUMA systems
  • Heterogeneous multiprocessing

Підходи до Multiple-Processor Scheduling

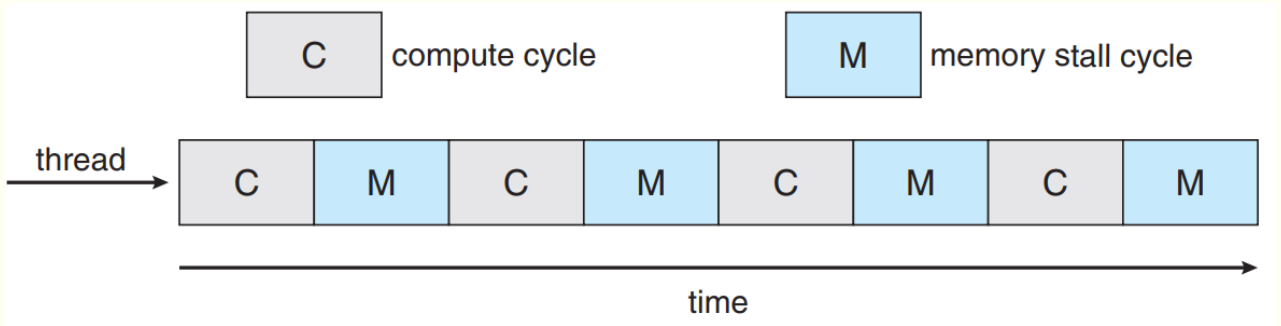


- One approach to CPU scheduling in a multiprocessor system has all scheduling decisions, I/O processing, and other system activities handled by a single processor — the master server.
  - The other processors execute only user code.
  - This **asymmetric multiprocessing** is simple because only one core accesses the system data structures, reducing the need for data sharing.
  - The downfall of this approach is the master server becomes a potential bottleneck where overall system performance may be reduced.
- The standard approach for supporting multiprocessors is **symmetric multiprocessing** (**SMP**), where each processor is self-scheduling.
  - Scheduling proceeds by having the scheduler for each processor examine the ready queue and select a thread to run.
  - Note that this provides two possible strategies for organizing the threads eligible to be scheduled:
    1. All threads may be in a common ready queue.
    2. Each processor may have its own private queue of threads.

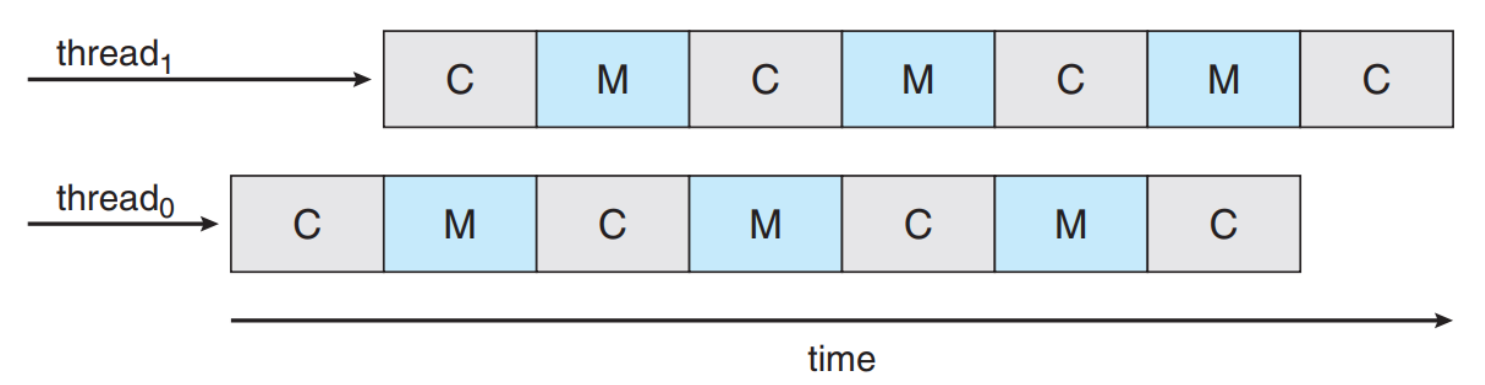
- If we select the first option, we have a possible race condition on the shared ready queue and therefore must ensure that two separate processors do not choose to schedule the same thread and that threads are not lost from the queue.
  - As discussed in Chapter 6, we could use some form of locking to protect the common ready queue from this race condition.
  - Locking would be highly contended, however, as all accesses to the queue would require lock ownership, and accessing the shared queue would likely be a performance bottleneck.
  - The second option permits each processor to schedule threads from its private run queue and therefore does not suffer from the possible performance problems associated with a shared run queue.
  - Thus, it is the most common approach on systems supporting SMP.
  - Additionally, as described in Section 5.5.4, having private, perprocessor run queues in fact may lead to more efficient use of cache memory.
  - There are issues with per-processor run queues—most notably, workloads of varying sizes.
  - However, as we shall see, balancing algorithms can be used to equalize workloads among all processors.
- Virtually all modern operating systems support SMP, including Windows, Linux, and macOS as well as mobile systems including Android and iOS.
  - In the remainder of this section, we discuss issues concerning SMP systems when designing CPU scheduling algorithms.

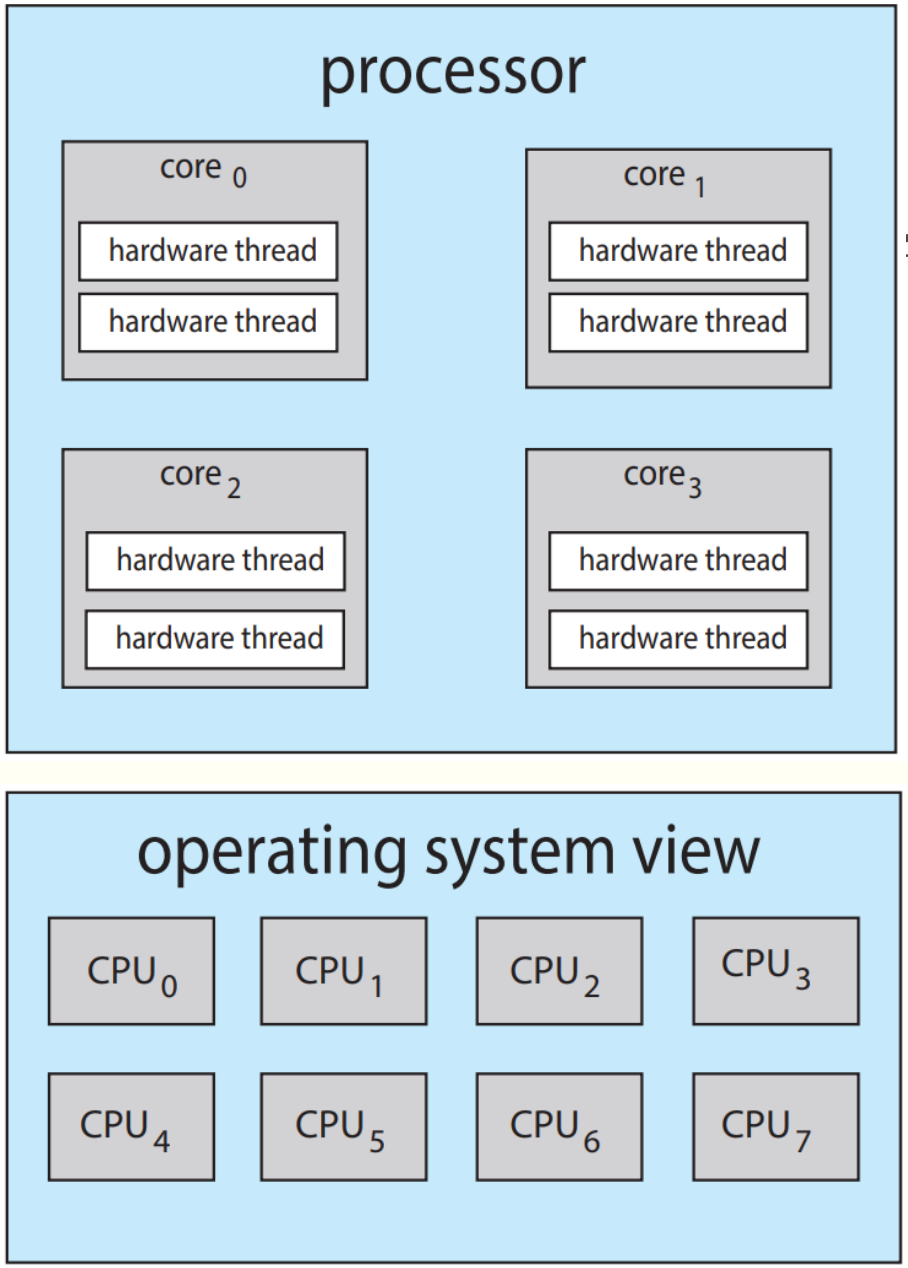
Multicore Processors

- Traditionally, SMP systems have allowed several processes to run in parallel by providing multiple physical processors.
  - However, most contemporary computer hardware now places multiple computing cores on the same physical chip, resulting in a **multicore processor**.
  - Each core maintains its architectural state and thus appears to the operating system to be a separate logical CPU.
  - SMP systems that use multicore processors are faster and consume less power than systems in which each CPU has its own physical chip.
- Multicore processors may complicate scheduling issues.
  - Let's consider how this can happen. Researchers have discovered that when a processor accesses memory, it spends a significant amount of time waiting for the data to become available.
  - This situation, known as a **memory stall**, occurs primarily because modern processors operate at much faster speeds than memory.
  - However, a memory stall can also occur because of a cache miss (accessing data that are not in cache memory).
  - In this scenario, the processor can spend up to 50 percent of its time waiting for data to become available from memory.



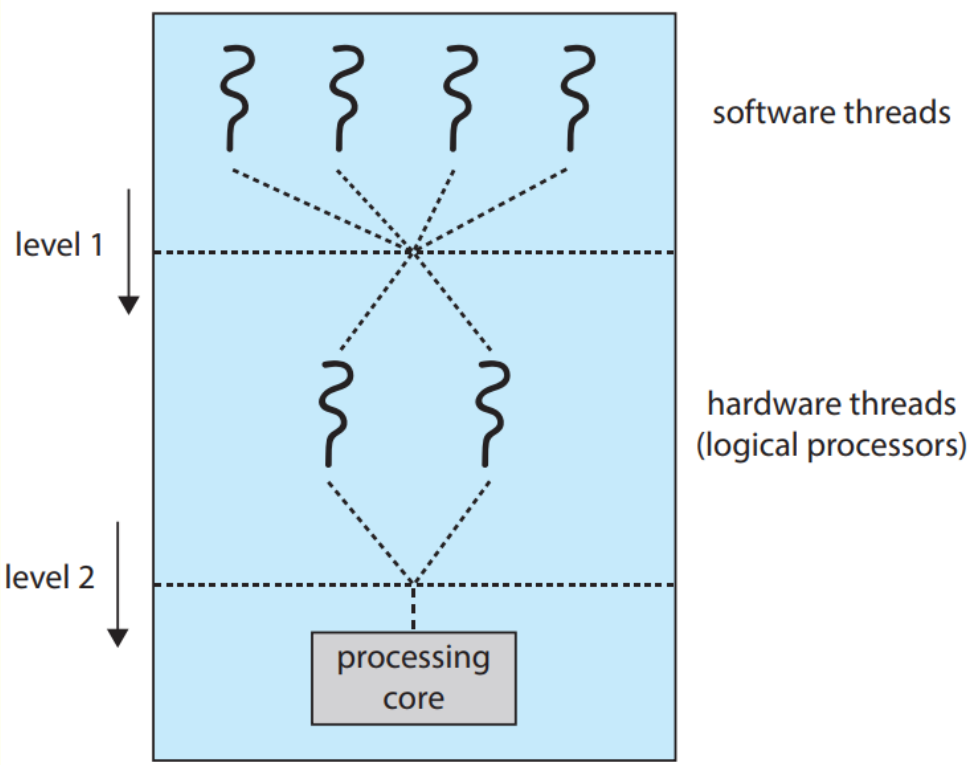
- To remedy this situation, many recent hardware designs have implemented multithreaded processing cores in which two (or more) **hardware threads** are assigned to each core.
  - That way, if one hardware thread stalls while waiting for memory, the core can switch to another thread.
  - Figure 5.13 illustrates a dual-threaded processing core on which the execution of thread 0 and the execution of thread 1 are interleaved.





- From an operating system perspective, each hardware thread maintains its architectural state, such as instruction pointer and register set, and thus appears as a logical CPU that is available to run a software thread.
  - This technique—known as **chip multithreading** (CMT).
  - Here, the processor contains four computing cores, with each core containing two hardware threads.
  - From the perspective of the operating system, there are eight logical CPUs.
- Intel processors use the term **hyper-threading** (also known as **simultaneous multithreading** or SMT) to describe assigning multiple hardware threads to a single processing core.
  - Contemporary Intel processors—such as the i7—support two threads per core, while the Oracle Sparc M7 processor supports eight threads per core, with eight cores per processor, thus providing the operating system with 64 logical CPUs.

Два рівні планування



In general, there are two ways to multithread a processing core: **coarsegrained** and **fine-graine** multithreading.

- With coarse-grained multithreading, a thread executes on a core until a long-latency event such as a memory stall occurs.
- Because of the delay caused by the long-latency event, the core must switch to another thread to begin execution.
- However, the cost of switching between threads is high, since the instruction pipeline must be flushed before the other thread can begin execution on the processor core.
- Once this new thread begins execution, it begins filling the pipeline with its instructions.
- Fine-grained (or interleaved) multithreading switches between threads at a much finer level of granularity—typically at the boundary of an instruction cycle.
- However, the architectural design of fine-grained systems includes logic for thread switching.
- As a result, the cost of switching between threads is small.

It is important to note that the resources of the physical core (such as caches and pipelines) must be shared among its hardware threads, and therefore a processing core can only execute one hardware thread at a time.

- Consequently, a multithreaded, multicore processor actually requires two different levels of scheduling, which illustrates a dual-threaded processing core.

Два рівні планування

- On one level are the scheduling decisions that must be made by the operating system as it chooses which software thread to run on each hardware thread (logical CPU).
  - For all practical purposes, such decisions have been the primary focus of this chapter.
  - Therefore, for this level of scheduling, the operating system may choose any scheduling algorithm, including those described in Section 5.3.
- A second level of scheduling specifies how each core decides which hardware thread to run.
  - There are several strategies to adopt in this situation.
  - One approach is to use a simple round-robin algorithm to schedule a hardware thread to the processing core.
  - This is the approach adopted by the UltraSPARC T3.
  - Another approach is used by the Intel Itanium, a dual-core processor with two hardware-managed threads per core.
  - Assigned to each hardware thread is a dynamic ***urgency*** value ranging from 0 to 7, with 0 representing the lowest urgency and 7 the highest.
  - The Itanium identifies five different events that may trigger a thread switch.
  - When one of these events occurs, the thread-switching logic compares the urgency of the two threads and selects the thread with the highest urgency value to execute on the processor core.
- Note that the two different levels of scheduling shown in Figure 5.15 are not necessarily mutually exclusive.
  - In fact, if the operating system scheduler (the first level) is made aware of the sharing of processor resources, it can make more effective scheduling decisions.
  - As an example, assume that a CPU has two processing cores, and each core has two hardware threads.
  - If two software threads are running on this system, they can be running either on the same core or on separate cores.
  - If they are both scheduled to run on the same core, they have to share processor resources and thus are likely to proceed more slowly than if they were scheduled on separate cores.
  - If the operating system is aware of the level of processor resource sharing, it can schedule software threads onto logical processors that do not share resources.

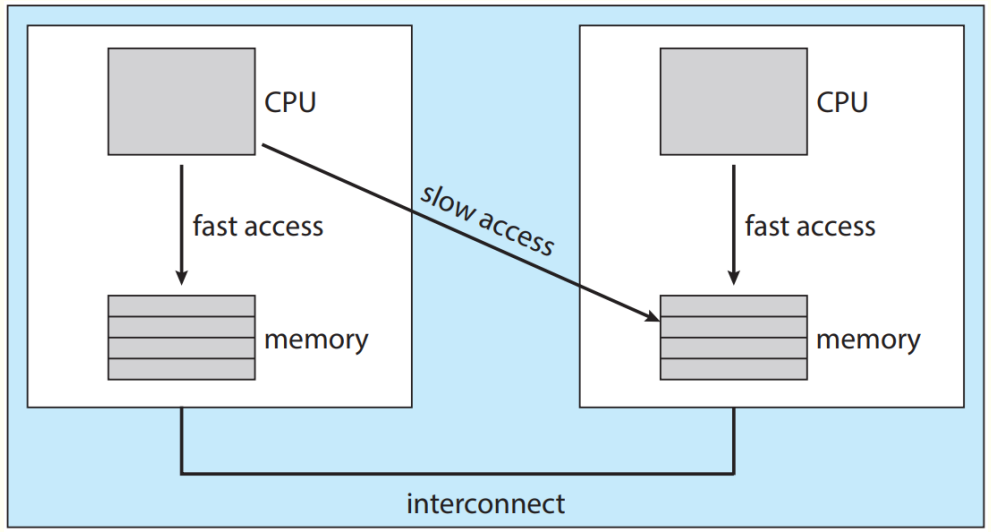
Load Balancing

- On SMP systems, it is important to keep the workload balanced among all processors to fully utilize the benefits of having more than one processor.
  - Otherwise, one or more processors may sit idle while other processors have high workloads, along with ready queues of threads awaiting the CPU.
  - **Load balancing** attempts to keep the workload evenly distributed across all processors in an SMP system. It is important to note that load balancing is typically necessary only on systems where each processor has its own private ready queue of eligible threads to execute.
  - On systems with a common run queue, load balancing is unnecessary, because once a processor becomes idle, it immediately extracts a runnable thread from the common ready queue.
- There are two general approaches to load balancing: push migration and pull migration.
  - With **push migration**, a specific task periodically checks the load on each processor and—if it finds an imbalance—evenly distributes the load by moving (or pushing) threads from overloaded to idle or less-busy processors.
  - **Pull migration** occurs when an idle processor pulls a waiting task from a busy processor. Push and pull migration need not be mutually exclusive and are, in fact, often implemented in parallel on load-balancing systems.
  - For example, the Linux CFS scheduler and the ULE scheduler available for FreeBSD systems implement both techniques.
- The concept of a “balanced load” may have different meanings.
  - One view of a balanced load may require simply that all queues have approximately the same number of threads.
  - Alternatively, balance may require an equal distribution of thread priorities across all queues.
  - In addition, in certain situations, neither of these strategies may be sufficient. Indeed, they may work against the goals of the scheduling algorithm.

Processor Affinity

- Consider what happens to cache memory when a thread has been running on a specific processor.
  - The data most recently accessed by the thread populate the cache for the processor.
  - As a result, successive memory accesses by the thread are often satisfied in cache memory (known as a “warm cache”).
  - Now consider what happens if the thread migrates to another processor—say, due to load balancing.
  - The contents of cache memory must be invalidated for the first processor, and the cache for the second processor must be repopulated.
  - Because of the high cost of invalidating and repopulating caches, most operating systems with SMP support try to avoid migrating a thread from one processor to another and instead attempt to keep a thread running on the same processor and take advantage of a warm cache.
  - This is known as **processor affinity** —that is, a process has an affinity for the processor on which it is currently running.
- The two strategies for organizing the queue of threads available for scheduling have implications for processor affinity.
  - If we adopt the approach of a common ready queue, a thread may be selected for execution by any processor.
  - Thus, if a thread is scheduled on a new processor, that processor’s cache must be repopulated.
  - With private, per-processor ready queues, a thread is always scheduled on the same processor and can therefore benefit from the contents of a warm cache.
  - Essentially, per-processor ready queues provide processor affinity for free!
- Processor affinity takes several forms.
  - When an operating system has a policy of attempting to keep a process running on the same processor—but not guaranteeing that it will do so—we have a situation known as **soft affinity**.
  - Here, the operating system will attempt to keep a process on a single processor, but it is possible for a process to migrate between processors during load balancing.
  - In contrast, some systems provide system calls that support **hard affinity**, thereby allowing a process to specify a subset of processors on which it can run.
  - Many systems provide both soft and hard affinity. For example, Linux implements soft affinity, but it also provides the sched_setaffinity() system call, which supports hard affinity by allowing a thread to specify the set of CPUs on which it is eligible to run.

NUMA and CPU scheduling



- The main-memory architecture of a system can affect processor affinity issues as well.
  - Although a system interconnect allows all CPUs in a NUMA system to share one physical address space, a CPU has faster access to its local memory than to memory local to another CPU.
  - If the operating system's CPU scheduler and memory-placement algorithms are ***NUMA-aware*** and work together, then a thread that has been scheduled onto a particular CPU can be allocated memory closest to where the CPU resides, thus providing the thread the fastest possible memory access.
- Interestingly, load balancing often counteracts the benefits of processor affinity.
  - That is, the benefit of keeping a thread running on the same processor is that the thread can take advantage of its data being in that processor's cache memory.
  - Balancing loads by moving a thread from one processor to another removes this benefit.
  - Similarly, migrating a thread between processors may incur a penalty on NUMA systems, where a thread may be moved to a processor that requires longer memory access times.
  - In other words, there is a natural tension between load balancing and minimizing memory access times.
  - Thus, scheduling algorithms for modern multicore NUMA systems have become quite complex.

Heterogeneous Multiprocessing

- Although mobile systems now include multicore architectures, some systems are now designed using cores that run the same instruction set, yet vary in terms of their clock speed and power management, including the ability to adjust the power consumption of a core to the point of idling the core.
  - Such systems are known as **heterogeneous multiprocessing** (HMP).
  - Note this is not a form of asymmetric multiprocessing as described in Section 5.5.1 as both system and user tasks can run on any core.
  - Rather, the intention behind HMP is to better manage power consumption by assigning tasks to certain cores based upon the specific demands of the task.
- For ARM processors that support it, this type of architecture is known as **big.LITTLE** where higher-performance ***big*** cores are combined with energy efficient ***LITTLE*** cores.
  - ***Big*** cores consume greater energy and therefore should only be used for short periods of time. Likewise, ***little*** cores use less energy and can therefore be used for longer periods.
- There are several advantages to this approach.
  - By combining a number of slower cores with faster ones, a CPU scheduler can assign tasks that do not require high performance, but may need to run for longer periods, (such as background tasks) to little cores, thereby helping to preserve a battery charge.
  - Similarly, interactive applications which require more processing power, but may run for shorter durations, can be assigned to big cores.
  - Additionally, if the mobile device is in a power-saving mode, energy-intensive big cores can be disabled and the system can rely solely on energy-efficient little cores.
  - Windows 10 supports HMP scheduling by allowing a thread to select a scheduling policy that best supports its power management demands.