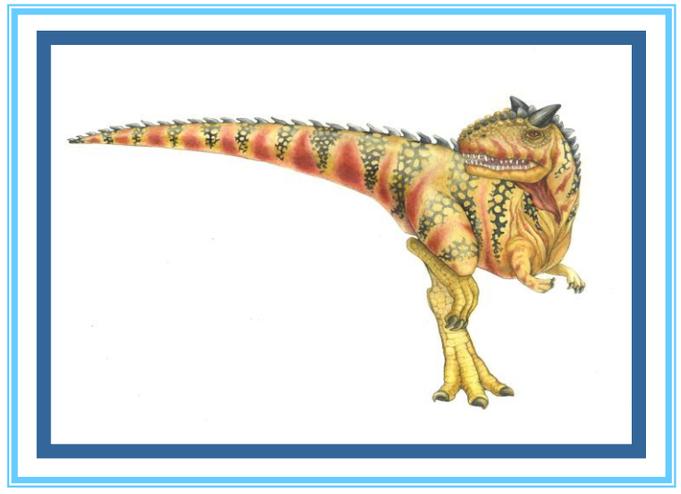
Chapter 6: CPU Scheduling



Silberschatz, Galvin and Gagne ©2013



Chapter 6: CPU Scheduling

- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Thread Scheduling
- Multiple-Processor Scheduling
- Real-Time CPU Scheduling
- Operating Systems Examples
- Algorithm Evaluation

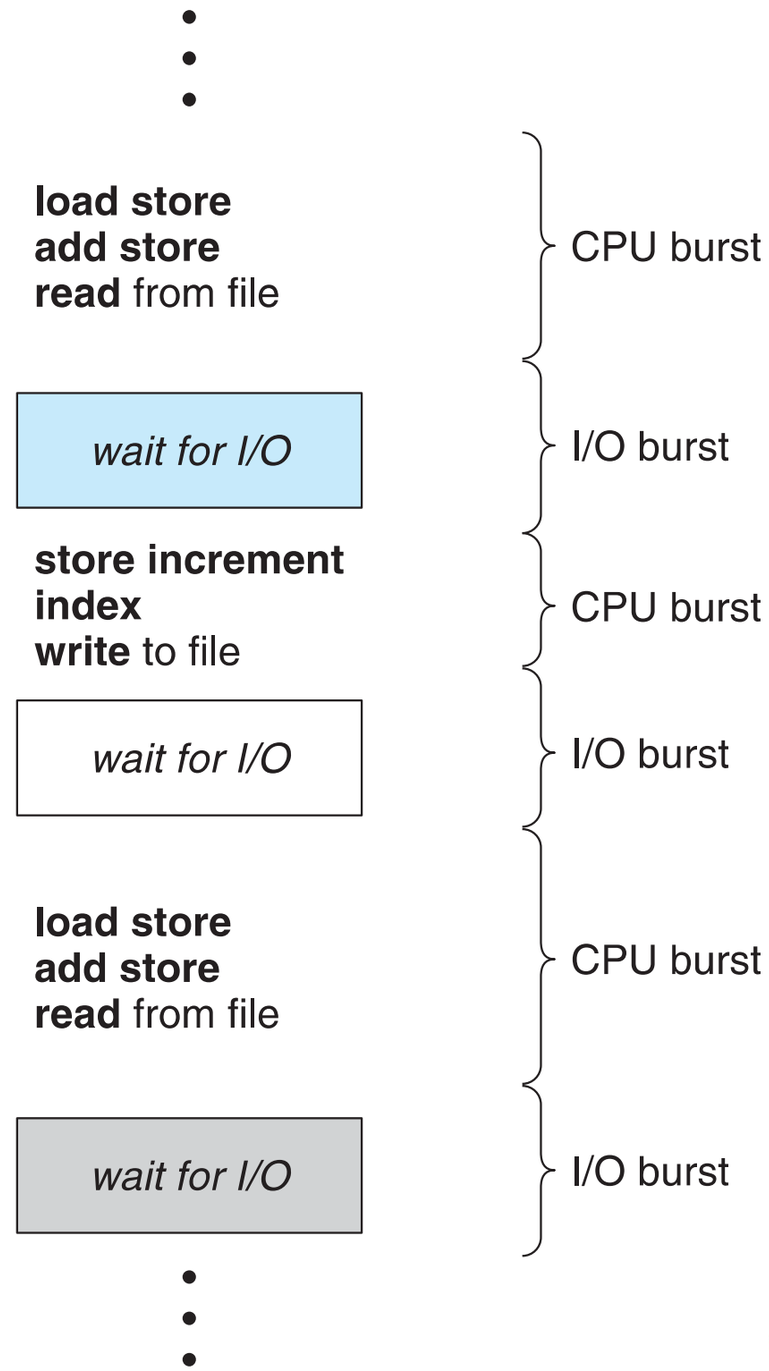


Basic Concepts

- Maximum CPU utilization obtained with multiprogramming
- CPU–I/O Burst Cycle – Process execution consists of a cycle of CPU execution and I/O wait
- CPU burst followed by I/O burst
- CPU burst distribution is of main concern

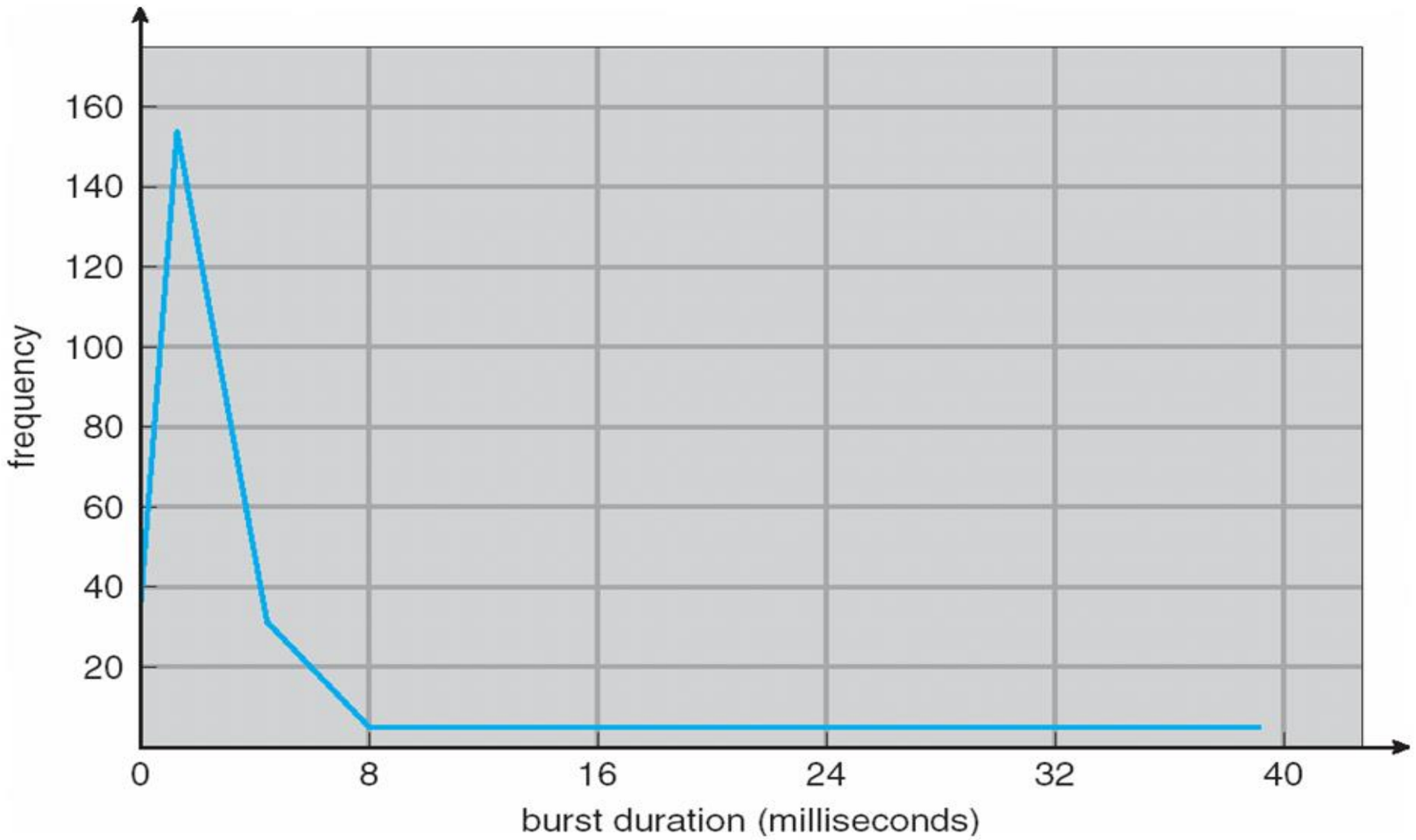
العمليات ما تشتغل بشكل مستمر
تمشي بدورة:
CPU → I/O → CPU → I/O
الهدف:
نخلي الـ CPU يكون شغال طول الوقت وما يضيع وقته
عني:
يكون عندنا أكثر من Process في الذاكرة
إذا واحدة وقفت (مثلاً تنتظر I/O)
نبدّل مباشرة لعملية ثانية

النتيجة:
أهم شيء نركز عليه:
مدة الـ CPU bursts





Histogram of CPU-burst Times







CPU Scheduler

مسمى آخر له

- **Short-term scheduler** selects from among the processes in ready queue, and allocates the CPU to one of them
  - Queue may be ordered in various ways
- CPU scheduling decisions may take place when a process:
  1. Switches from running to waiting state
  2. Switches from running to ready state
  3. Switches from waiting to ready
  4. Terminates
- Scheduling under 1 and 4 is **nonpreemptive**
- All other scheduling is **preemptive**
  - Consider access to shared data
  - Consider preemption while in kernel mode
  - Consider interrupts occurring during crucial OS activities

هو جزء من نظام التشغيل:
يختار أي Process من الـ Ready Queue يشتغل على الـ CPU

Ready Queue :
هي قائمة فيها العمليات الجاهزة للتنفيذ (تنتظر دورها على الـ CPU)

النظام يقرر يبدل العملية في 4 حالات:
1 من Running → Waiting
(مثلاً طلب I/O)
العملية توقفت بنفسها
2 من Running → Ready
تم سحب الـ CPU منها (انقطع عليها)
3 من Waiting → Ready
خلصت I/O ورجعت تنتظر دورها
4 Terminated
خلصت العملية نهائيًا

في الـ Preemptive لازم ننتبه لـ:
البيانات المشتركة (shared data)
المقاطعات (interrupts)
إذا النظام داخل kernel mode
لأن الإيقاف المفاجئ ممكن يسبب مشاكل





Dispatcher

- Dispatcher module gives control of the CPU to the process selected by the short-term scheduler; this involves:
  - switching context
  - switching to user mode
  - jumping to the proper location in the user program to restart that program

what is the function of dispatcher?

- **Dispatch latency** – time it takes for the dispatcher to stop one process and start another running

Dispatcher

💡 وظيفته؟

بعد ما الـ Scheduler يختار العملية:

هو اللي يعطيها الـ CPU فعليًا

🛠️ يسوي 3 أشياء:

1. Context Switching

يحفظ حالة العملية القديمة

ويرجع حالة العملية الجديدة

2. Switch to User Mode

يحول من:

Kernel mode → User mode

3. Jump to Program

يرجع يكمل تنفيذ البرنامج من المكان اللي وقف فيه

⏱️ Dispatch Latency

هو:

الوقت اللي يحتاجه النظام عشان يبدل من Process إلى Process

👉 كل ما كان أقل = النظام أسرع 🔥

الخلاصة السريعة:

🧠 Scheduler:

يقرر مين يشتغل

⚙️ Dispatcher:

ينفذ القرار فعليًا





Scheduling Criteria معايير التقييم

الهدف: يكون الـ CPU مشغول طول الوقت

- **CPU utilization** – keep the CPU as busy as possible

الانتاجية: كم عملية تخلص في وقت معين

- **Throughput** – # of processes that complete their execution per time unit

الوقت من بداية العملية إلى نهايتها

- **Turnaround time** – amount of time to execute a particular process

كم جلست العملية تنتظر في الـ Ready Queue

- **Waiting time** – amount of time a process has been waiting in the ready queue

- **Response time** – amount of time it takes from when a request was submitted until the first response is produced, not output (for time-sharing environment)

كم أخذ وقت أول ما استجاب النظام
⚠ مهم:
مو وقت إنهاء العملية
فقط أول استجابة





Scheduling Algorithm Optimization Criteria

- Max CPU utilization
- Max throughput
- Min turnaround time
- Min waiting time
- Min response time

CPU Utilization	🔼 maximize
Throughput	🔼 maximize
Turnaround Time	🔽 minimize
Waiting Time	🔽 minimize
Response Time	🔽 minimize





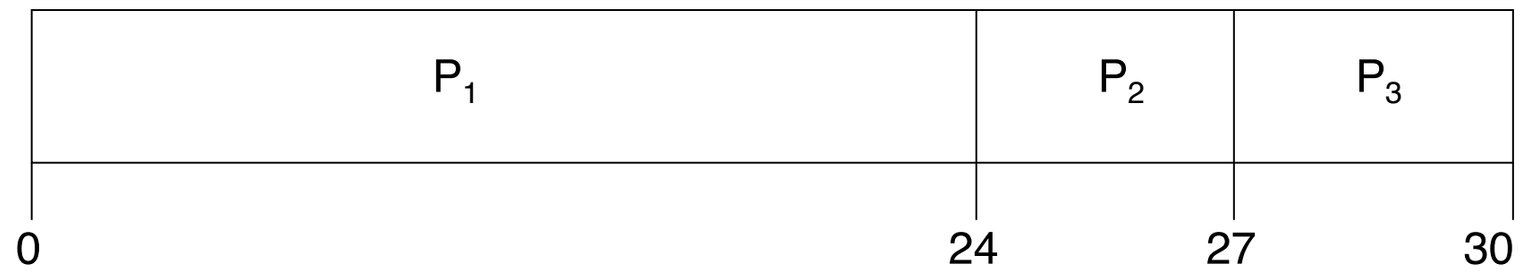
First-Come, First-Served (FCFS) Scheduling

أول Process يوصل ← أول واحد ينفذ

زي الطابور 👇

Process	Burst Time
P_1	24
P_2	3
P_3	3

- Suppose that the processes arrive in the order: P_1 , P_2 , P_3
  The Gantt Chart for the schedule is:



- Waiting time for P_1 = 0; P_2 = 24; P_3 = 27
- Average waiting time: (0 + 24 + 27)/3 = 17 سيء جدًا! ❌



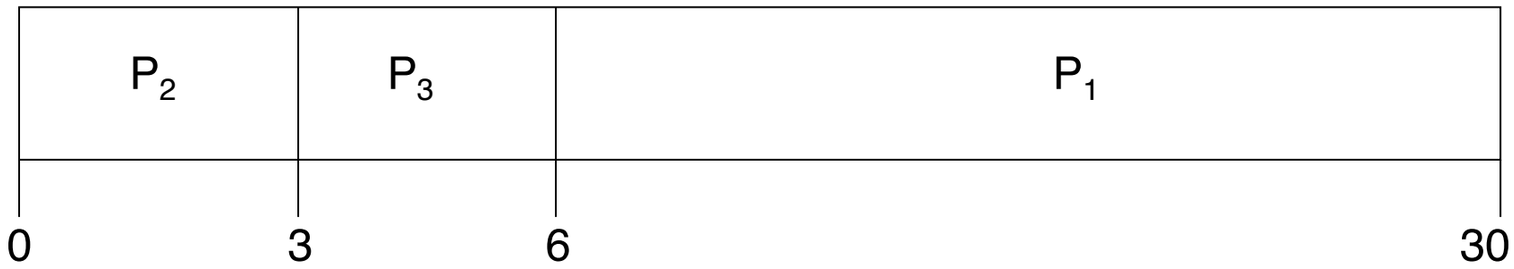


FCFS Scheduling (Cont.)

Suppose that the processes arrive in the order:

$$P_2 , P_3 , P_1$$

- The Gantt chart for the schedule is:



- Waiting time for $P_1 = 6; P_2 = 0, P_3 = 3$
- Average waiting time: (6 + 0 + 3)/3 = 3
- Much better than previous case أفضل بكثير! 🔥
- Convoy effect - short process behind long process
  - Consider one CPU-bound and many I/O-bound processes

❗ نفس العمليات... لكن ترتيب مختلف = نتيجة مختلفة جدًا
👉 هذا يبين إن FCFS:
يتأثر بشدة بترتيب الوصول





Shortest-Job-First (SJF) Scheduling

- Associate with each process the length of its next CPU burst
  - Use these lengths to schedule the process with the shortest time
- SJF is optimal – gives minimum average waiting time for a given set of processes
  - The difficulty is knowing the length of the next CPU request
  - Could ask the user

💡 الفكرة:
نختار العملية اللي عندها أقصر CPU burst
يعني:
الأقصر يشتغل أول
الأطول ينتظر
🎯 الهدف:
تقليل:
Waiting time
Turnaround time
👉 ولهذا:
SJF = Optimal (الأفضل رياضيًا)
⚠️ المشكلة:
كيف نعرف مدة الـ CPU burst القادمة؟
❌ صعب جدًا
(النظام ما يعرف المستقبل)
👉 أحيانًا:
يطلب من المستخدم (نادر)
أو يتوقعها (prediction)

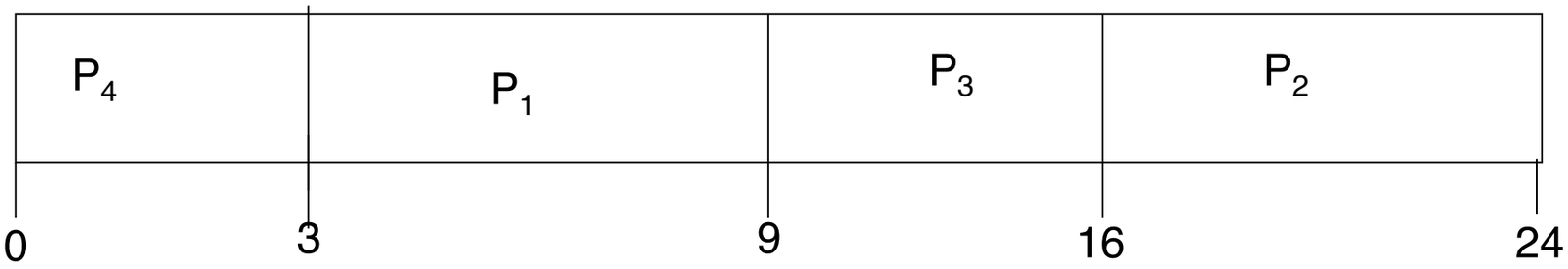




Example of SJF

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

- SJF scheduling chart



- Average waiting time = (3 + 16 + 9 + 0) / 4 = 7 (أقل من FCFS بكثير)





Shortest-Job-First (SJF) Scheduling

- Associate with each process the length of its next CPU burst. Use these lengths to schedule the process with the shortest time
- Two schemes:
  - nonpreemptive – once CPU given to the process it cannot be preempted until completes its CPU burst
  - preemptive – if a new process arrives with CPU burst length less than remaining time of current executing process, preempt. This scheme is know as the Shortest-Remaining-Time-First (SRTF)
- SJF is optimal – gives minimum average waiting time for a given set of processes

SJF نوعين من 🔴
1️⃣ Non-preemptive SJF
إذا بدأ Process → يكمل للنهاية
ما نقاطعه
2️⃣ Preemptive SJF (SRTF) ⭐
اسمه:
Shortest Remaining Time First
الفكرة: 💡
إذا جا Process جديد:
وكان أقصر من الباقي من العملية الحالية
نوقف الحالية ونبدله 👉

SJF تقييم ⚖️
المميزات: ✅
أقل Waiting Time (الأفضل نظريًا)
أداء ممتاز
العيوب: ❌
ما نعرف burst time
فيه Starvation
صعب التطبيق عمليًا

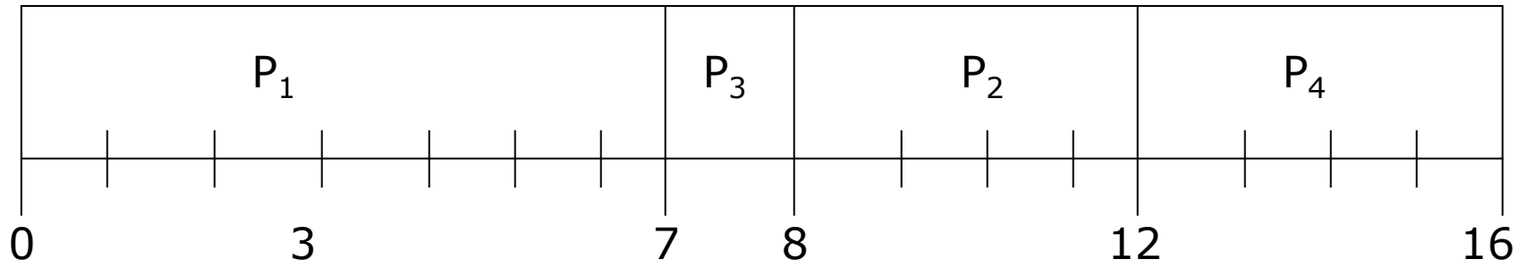




Example of Non-Preemptive SJF

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- SJF (non-preemptive)



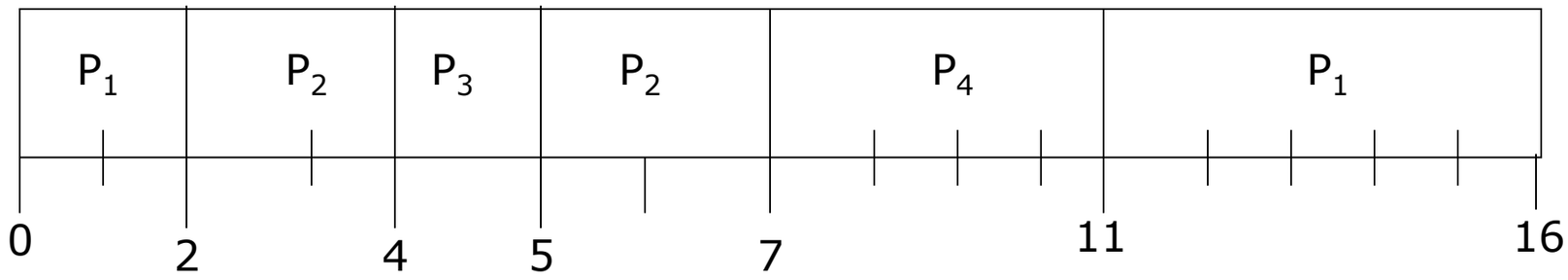
- Average waiting time = (p1+p2+p3+p4)
  = (0 + 6 + 3 + 7)/4
  = 4



Example of Preemptive SJF

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- SJF (preemptive)



- Average waiting time = (9 + 1 + 0 +2)/4 = 3 Non-preemptive أقل من 🔥

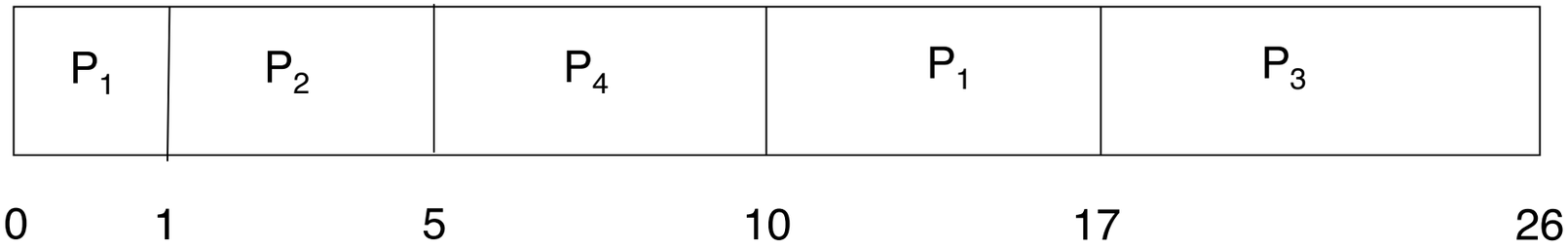


Additional Example

- Now we add the concepts of varying arrival times and preemption to the analysis

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

- *Preemptive* SJF Gantt Chart



- Average waiting time = [(10-1)+(1-1)+(17-2)+5-3)]/4 = 26/4 = 6.5 msec

🧠 الفكرة (باختصار):
يبدأ بـ P1
كل ما يجي Process أقصر → يقاطعه
دايم نختار أقل Remaining Time
📉 النتيجة:
Average Waiting Time = 6.5
🔥 الخلاصة النهائية:
🟡 SJF:
الأفضل نظريًا
يقلل الانتظار
🔴 SRTF:
نسخة أقوى
يعطي نتائج أفضل
لكن أعقد





Priority Scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer ≡ highest priority)
  - Preemptive
  - Nonpreemptive
- SJF is priority scheduling where priority is the inverse of predicted next CPU burst time
- Problem ≡ **Starvation** – low priority processes may never execute
- Solution ≡ **Aging** – as time progresses increase the priority of the process

💡 الفكرة:
كل Process لها Priority (أولوية)
والـ CPU يروح للي عنده أعلى أولوية
⚠️ في السلايد:
أصغر رقم = أعلى أولوية

🔁 نوعين:
1️⃣ Non-preemptive
إذا بدأت عملية → تكمل
حتى لو جاء Process أعلى أولوية

2️⃣ Preemptive
إذا جاء Process أعلى أولوية
👉 نوقف الحالية فورًا

SJF هو نوع من Priority Scheduling
لكن:
الأولوية = 1 / burst time
يعني: الأقصر → أعلى أولوية

⚠️ المشكلة: Starvation
💡 وش يصير؟ العمليات ذات الأولوية المنخفضة ممكن ما تشتغل أبدًا
ليش؟ كل شوي تجي عمليات أعلى أولوية 😬
🛠️ الحل: Aging
💡 الفكرة: كل ما انتظرت العملية أكثر → نرفع أولويتها
مثال:
كانت Priority = 5
بعد فترة تصير = 4 → 3 → 2
👉 بالنهاية راح تشتغل



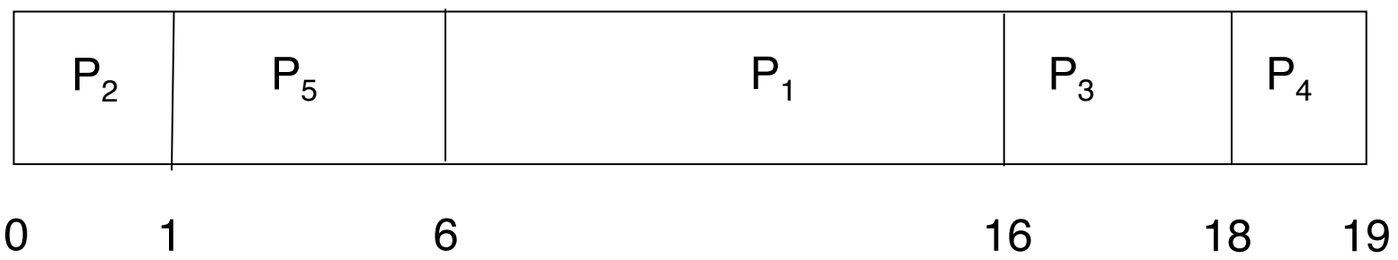


Example of Priority Scheduling

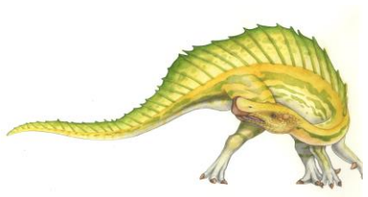
Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

✅ المميزات:
مرن (نقدر نتحكم بالأولوية)
مناسب للأنظمة المهمة (real-time جزئيًا)
❌ العيوب:
Starvation 😬
يحتاج إدارة للأولويات

- Priority scheduling Gantt Chart



- Average waiting time = (P2+ …. +p4) = 8.2 msec





Round Robin (RR)

- Each process gets a small unit of CPU time (time quantum q), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue.
- 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 at once. No process waits more than $(n-1)q$ time units.
- Timer interrupts every quantum to schedule next process
- Performance
  - q large $\Rightarrow$ FIFO
  - q small $\Rightarrow$ q must be large with respect to context switch, otherwise overhead is too high

💡 الفكرة:
كل Process يأخذ وقت صغير ثابت (Time Quantum = q)
وبعده:
👉 نوقفه (preempt)
👉 ونحطه آخر الـ Ready Queue
زي طابور دائري 🎡
كل واحد يأخذ دوره شوي ويرجع ورا

⏱️ Time Quantum (q)
عادة: 10 – 100 ms
هو أقصى وقت لكل Process
🔁 كيف تشتغل؟
نعطي Process وقت = q
إذا خلص قبل q → يطلع
إذا ما خلص → نوقفه ونرجعه آخر الطابور

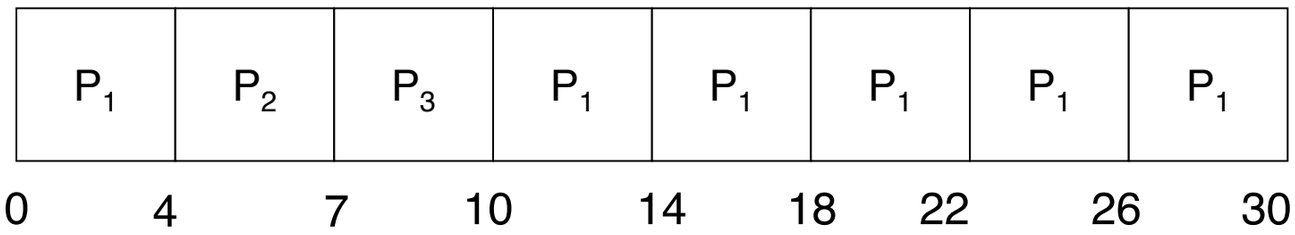




Example of RR with Time Quantum = 4

Process	Burst Time
P_1	24
P_2	3
P_3	3

- The Gantt chart is:

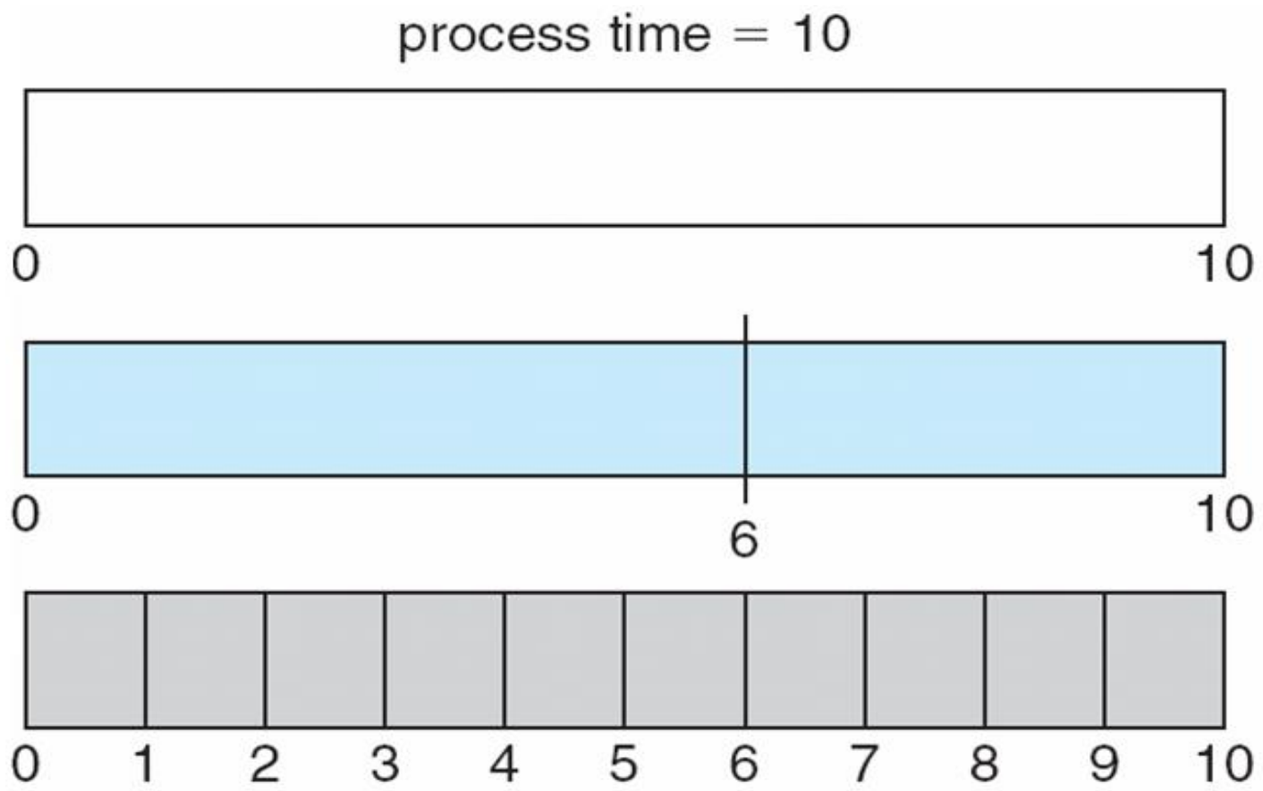


- Typically, higher average turnaround than SJF, but better ***response***
- q should be large compared to context switch time
- q usually 10ms to 100ms, context switch < 10 usec

⏱️ 0 → 4
أخذ 4 (باقي 20) P1
⏱️ 4 → 7
يخلص (3) P2
⏱️ 7 → 10
يخلص (3) P3
بعدها:
نرجع لـ P1 ونكمله كل مرة 4
💥 الفكرة المهمة:
كل Process يأخذ فرصة بسرعة
حتى لو كان طويل
⚖️ تأثير قيمة q
🟢 إذا q كبير:
👉 RR ≈ FCFS
(لأن كل Process يخلص بدوره)
🔴 إذا q صغير جدًا:
👉 مشاكل:
😬 كثير Context Switching
النظام يصير بطيء



Time Quantum and Context Switch Time



quantum	context switches
12	0
6	1
1	9



Multilevel Queue

- Ready queue is partitioned into separate queues, eg:
  - foreground (interactive)
  - background (batch)
- Process permanently in a given queue
- Each queue has its own scheduling algorithm:
  - foreground – RR
  - background – FCFS
- Scheduling must be done between the queues:
  - Fixed priority scheduling; (i.e., serve all from foreground then from background). Possibility of starvation.
  - Time slice – each queue gets a certain amount of CPU time which it can schedule amongst its processes; i.e., 80% to foreground in RR
  - 20% to background in FCFS

💡 الفكرة:
بدل ما يكون عندنا Ready Queue واحدة
👉 نقسمها إلى عدة Queues
🧠 مثال:
🎮 Foreground (عمليات تفاعلية – user)
⚙️ Background (عمليات ثقيلة – batch)
كل نوع له Queue خاصة فيه

🚫 نقطة مهمة:
العملية تبقى في نفس الـ Queue طول حياتها
(ما تنتقل بين queues)

🔄 كيف نختار بين الـ Queues؟ فيه طريقتين:
1️⃣ Fixed Priority Scheduling
لها أولوية أعلى من الثانية Queue
مثلاً: Foreground أول دائمًا, بعده Background
⚠️ المشكلة: Starvation
الـ Background ممكن ما يشتغل أبدًا
2️⃣ Time Slicing بين Queues
نعطي كل Queue نسبة من وقت الـ CPU
مثلاً:
80% → Foreground
20% → Background
كل Queue تدير وقتها بنفسها

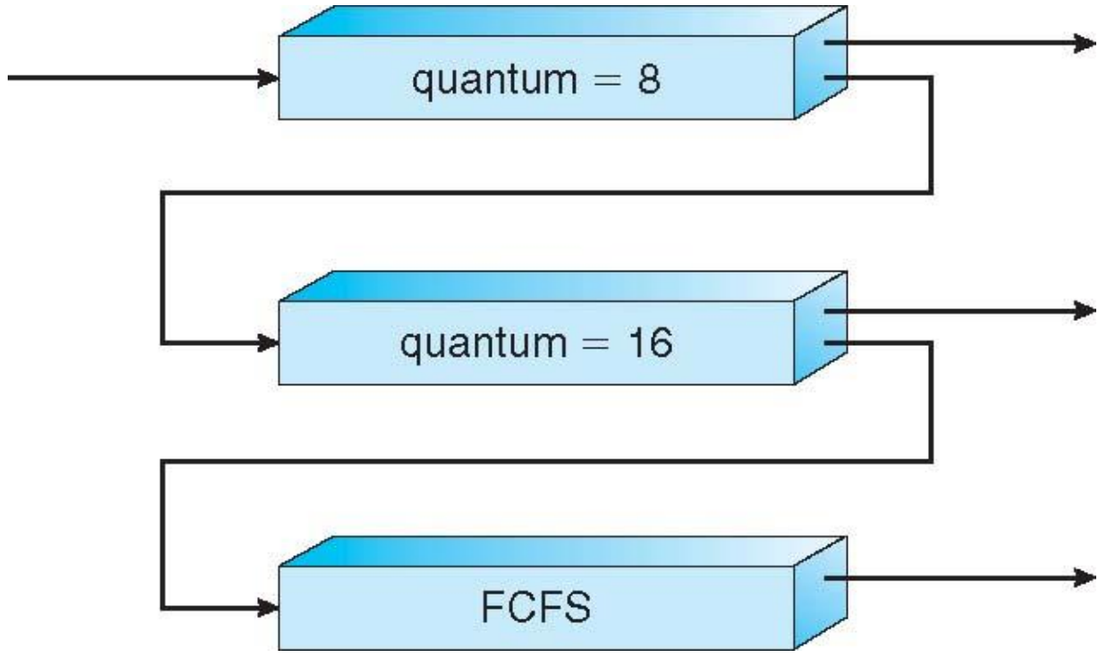




Example of Multilevel Feedback Queue

تقسيم العمليات حسب النوع = Multilevel Queue
وكل نوع له طريقة جدولة مختلفة

- Three queues:
  - Q_0 – RR with time quantum 8 milliseconds
  - Q_1 – RR time quantum 16 milliseconds
  - Q_2 – FCFS
- Scheduling
  - A new job enters queue Q_0 which is served FCFS
    - When it gains CPU, job receives 8 milliseconds
    - If it does not finish in 8 milliseconds, job is moved to queue Q_1
  - At Q_1 job is again served FCFS and receives 16 additional milliseconds
    - If it still does not complete, it is preempted and moved to queue Q_2







Linux Scheduling Through Version 2.5

- Prior to kernel version 2.5, ran variation of standard UNIX scheduling algorithm
- Version 2.5 moved to constant order *O*(1) scheduling time
  - SJF Preemptive, priority based
  - Two priority ranges: time-sharing and real-time
  - **Real-time** range from 0 to 99 and **nice** value from 100 to 140
  - Map into global priority with numerically lower values indicating higher priority
  - Higher priority gets larger q
  - Task run-able as long as time left in time slice (**active**)
  - If no time left (**expired**), not run-able until all other tasks use their slices
  - All run-able tasks tracked in per-CPU **runqueue** data structure
    - Two priority arrays (active, expired)
    - Tasks indexed by priority
    - When no more active, arrays are exchanged
  - Worked well, but poor response times for interactive processes

🐧 Linux (2.5 – قديم)
اختيار سريع ثابت → O(1)
يعتمد على priority + time slice
فيه:
Active (يشتغل)
Expired (ينتظر)
❌ مشكلته: ضعيف مع الـ interactive





Linux Scheduling in Version 2.6.23 +

Linux (حديث – CFS) 🐧
يعتمد على العدالة (fairness)
يستخدم vruntime
يختار أقل vruntime 👉
(إلى +19 20-) nice عبر priority
ما فيه quantum ثابت ❌

- ***Completely Fair Scheduler*** (CFS)
- **Scheduling classes**
  - Each has specific priority
  - Scheduler picks highest priority task in highest scheduling class
  - Rather than quantum based on fixed time allotments, based on proportion of CPU time
  - 2 scheduling classes included, others can be added
    1. default
    2. real-time
- Quantum calculated based on **nice value** from -20 to +19
  - Lower value is higher priority
  - Calculates **target latency** – interval of time during which task should run at least once
  - Target latency can increase if say number of active tasks increases
- CFS scheduler maintains per task **virtual run time** in variable **`vruntime`**
  - Associated with decay factor based on priority of task – lower priority is higher decay rate
  - Normal default priority yields virtual run time = actual run time
- To decide next task to run, scheduler picks task with lowest virtual run time





Windows Scheduling

- Windows uses priority-based preemptive scheduling
- Highest-priority thread runs next
- **Dispatcher** is scheduler (Merge)
- Thread runs until (1) blocks, (2) uses time slice, (3) preempted by higher-priority thread
- Real-time threads can preempt non-real-time
- 32-level priority scheme
- **Variable class** is 1-15, **real-time class** is 16-31
- Priority 0 is memory-management thread
- Queue for each priority
- If no run-able thread, runs **idle thread**

Windows
Priority-based + Preemptive
أعلى priority يشتغل
فيه 32 مستوى (0–31)
أعلى من العادي real-time
يوقف إذا:
خلص الوقت
جاء أعلى منه
طلب I/O

🔥 الخلاصة:
القديم Linux → سريع
الجديد Linux → عادل
Windows → يعتمد على الأولوية

