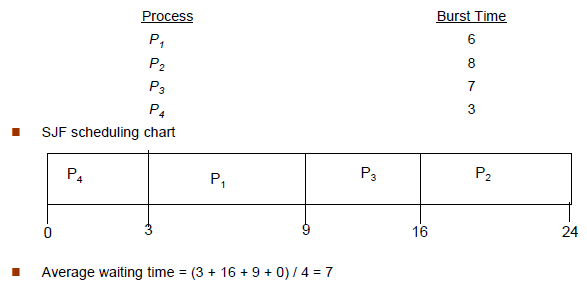
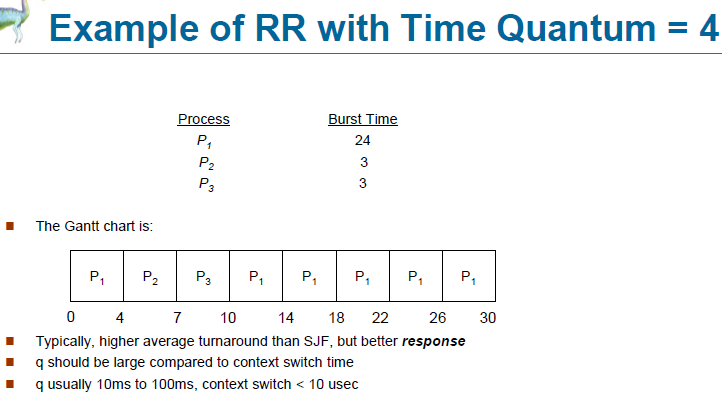
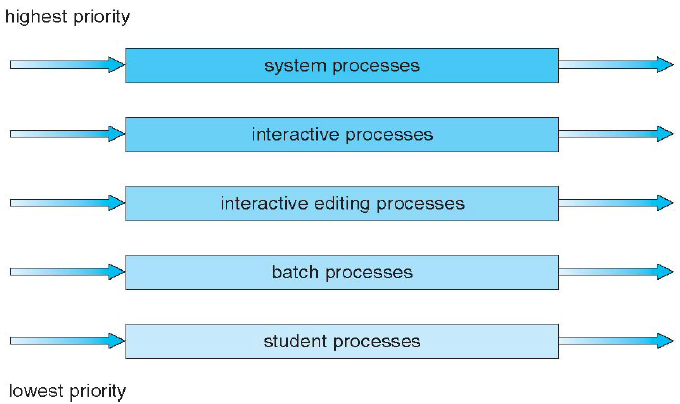
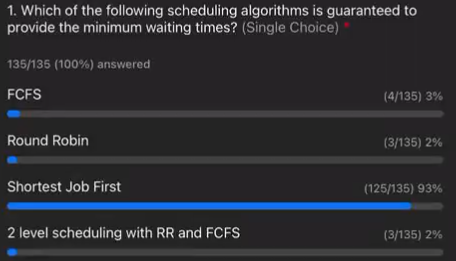
**Process Scheduling**

* 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
* Short-term scheduler, selects among proccesses in ready queue
* **Nonpreemptive:** Scheduling under normal actions like switcb from waiting to ready or else
* **Preemptive:** Shared data, preemption while in kernel mode, interrupts during cruical OS activities.
* **Dispatcher:** Gives control to process for switching context, to user mode, jump.
* **Scheduling Criterias:**
  + **Cpu utilization:** Keep CPU busy as possible.
  + **Throughtpu:** Number of procces that complete their execution
  + **Turnaround time:** Amount of time to execute a particular process.
  + **Waiting time**
  + Response time
* **First-Come, First-Served (FCFS) Scheduling**
  + metin içeren bir resim

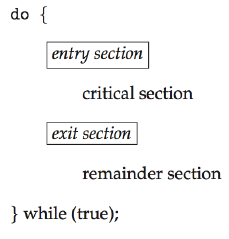
    Açıklama otomatik olarak oluşturuldu
  + **Convoy Effect:** Short process behind long process.
* **Shortest-Job-First (SJF) Scheduling**
  + 
  + metin içeren bir resim

    Açıklama otomatik olarak oluşturuldu
* **Shortest-remaining-time-first**
  + tablo içeren bir resim

    Açıklama otomatik olarak oluşturuldu
* **Round Robin (RR)**
  + 
* **Multilevel Queue**
  + Ready queue parts:
    - Foreground (interactive) 🡪 RR
    - Background (batch) 🡪 FCFS
  + 
* **Thread Scheduling**
  + Distinction between user-level and kernel-level threads
  + User-level 🡪 process-contention-scope (PCS)
  + Kernel-level 🡪 system-contention-scope (SCS)
* **Multiple-Processor Scheduling**
  + Homogeneous processors
  + **Asymmetric Multiprocessing:** Only one processor is self-scherduling, others are ready
  + **Symmetric Multiprocessing (SMP):** Each processor is self-scheduling, all processors are ready
  + **Processor affinity:** Soft, hard
* **Real-Time CPU Scheduling**
  + **Soft:** No guarantee
  + **Hard:** Task must be serviced by its deadline



**Synchronization**

* **Critical Section:** Each process has this. When one process in critical section, no other may be. Critical section problem is a design to solve this.
* 
* **Solution to Critical-Section Problem**
  + 1. Mutual Exclusion: If a P is executing their critical sect., then no other processes can.
  + 2. Progress: If no process exec. their c.s. and there exist a process that wants to enter their c.s., then cannot be postponed indefinetely.
  + 3. Bounded Waiting: Bounded functions
* **Peterson’s Solution**
  + Two process solution, sharing two variables: int turn, Boolean flag[2]
  + The variable turn indicates whose turn it is to enter the c.s.
  + The flag array is used to indicate if a process is ready. If flag[i] = true, ready.
  + metin içeren bir resim

    Açıklama otomatik olarak oluşturuldu
* **Locking:** Protecting critical regions
* **Atomic:** Non-interruptible
* **Semaphore**
  + Synchronization tool that does not require busy waiting.
  + Semaphore S 🡪 int value
  + Two operations: wait() and signal() (P() and V())
  + **Counting Semaphore:** int value
  + **Binary Semaphore:** int value can range only between 0 and 1
* **Deadlock:** Multiple processes are waiting indefinetely
* **Starvation:** A process may never be removed from the semaphore queue (indefinite blocking)
* **Priority Inversion:** Scheduling problem when lower-priority process holds a lock needed by higher priority process.
* Problems of Synchronization

Page and Frame Replacement Algorithms

Deadlock