Processes & Threads

Part I:Processes & Scheduling

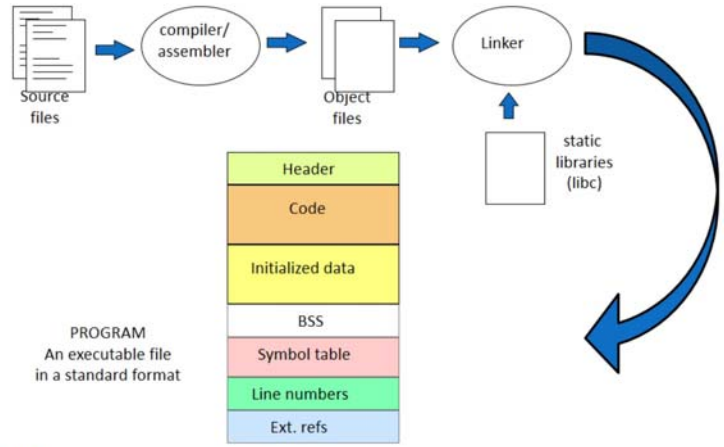
What is a program?

- A program is a file containing
  - **executable code** (machine instructions) and
  - **data** (information manipulated by these instructions)
  - together describe a computation
- Resides on disk
- Obtained through compilation and linking

Topics

- Processes
- Threads
- Scheduling
- Interprocess communication
- Classical IPC problems

Preparing a Program



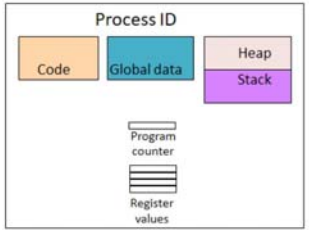
Running a program

- Every OS provides a "*loader*" that is capable of converting a given program into an executing instance, a process
  - A program in execution is called a process
- The loader:
  - reads and interprets the executable file
  - Allocates memory for the new process and sets process's memory to contain code & data from executable
  - pushes "argc", "argv", "envp" on the stack
  - sets the CPU registers properly & jumps to the entry point

What Is A Process?

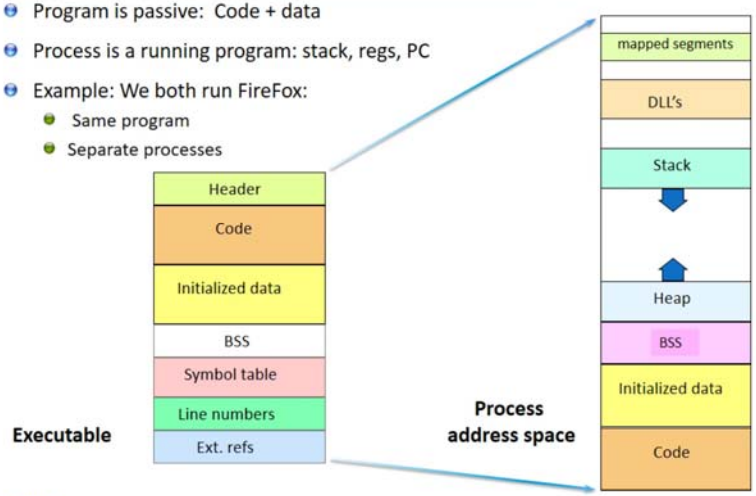
An operating system abstraction to represent what is needed to run a single program

- How are processes implemented in an operating system?
- A process is the basic unit of execution
  - it's the unit of scheduling
  - it's the dynamic (active) execution context (as opposed to a program, which is static)
- A process is sometimes called a job or a task
- Structure of a typical process:
  - Process identifier: unique ID of a process.
  - Process state: current activity of a process.
  - Process context: program counter (*PC*), register values, stack pointer (*SP*).
  - Memory: program text (i.e., code), global data, stack, and heap.



Process != Program

- Program is passive: Code + data
- Process is a running program: stack, regs, PC
- Example: We both run FireFox:
  - Same program
  - Separate processes



When is a process created?

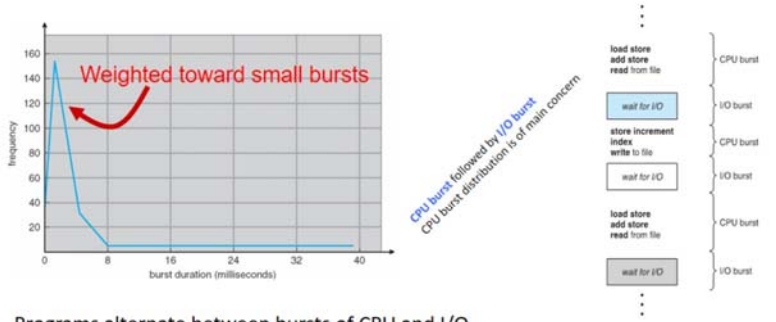
- Processes can be created in two ways
  - **System initialization:** one or more processes created when the OS starts up
  - **Execution of a process creation system call**: something explicitly asks for a new process
- System calls can come from
  - User request to create a new process *(system call executed from user shell)*
  - Already running processes
    - User programs
    - System daemons

When do processes end?

- Conditions that terminate processes can be
- Voluntary
  - Process executes last statement and calls exit syscall
    - Process' resources are deallocated by operating system
  - Error exit
    - Based on a certain condition *(file not found, etc.)*
    - Handled exception *(try…catch)*
- Involuntary
  - Parent may terminate execution of child process (kill)
    - Child has exceeded allocated resources
    - Task assigned to child is no longer required
    - If parent is exiting
      - Some OSes don't allow child to continue if parent terminates
        - All children terminated - cascading termination
  - Fatal error *(only sort of involuntary)*
    - Executing an illegal instruction
    - Accessing illegal memory addresses
  - Killed by another process

A. Elkady

Process Behavior



Programs alternate between bursts of CPU and I/O

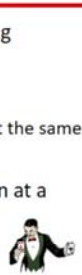
- Program typically uses the CPU for some period of time, then does I/O, then uses CPU again
- Each scheduling decision is about which job to give to the CPU for use by its next CPU burst

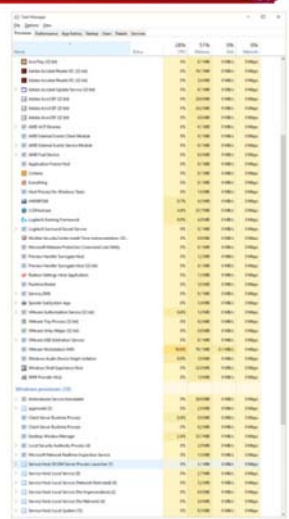
A. Elkady

The Process Model

Typically, a modern OS has multiple running processes

- But…..
  - How are these processes are running at the same time?
- It's an Illusion. Only one process can run at a time on the CPU
- Our illusionist's name is: *the scheduler*

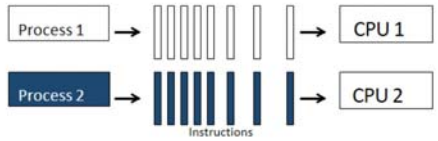




A. Elkady

Multiprocessing

- Many of today's computations can take advantage of multiple processing units (multi-core processors)

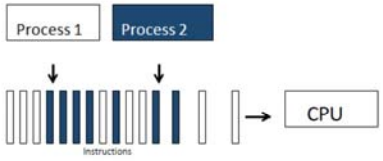


- Multiprocessing: the use of more than one processing unit in a system
- Execution of processes is said to be parallel, as they are running at the same time

A. Elkady 13

Multitasking/Multithreading

- Even on systems with a single processing unit we may give the illusion of that several programs run at once

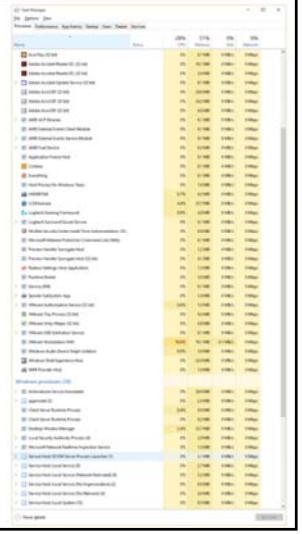


- Multitasking/multithreading: the operating system switches between the execution of different tasks/threads
- Execution of processes is said to be interleaved, as all are in progress, but only one is running at a time

A. Elkady

Process Management

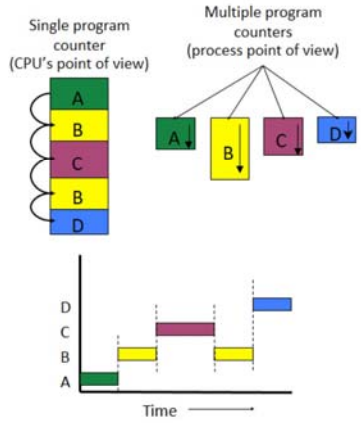
- Running Many Processes ???
- We have the basic mechanism to
  - switch between user processes and the kernel,
  - the kernel can switch among user processes,
  - Protect OS from user processes and processes from each other
- Process management deals with several issues:
  - How do we decide which user process to run?
  - How do we represent user processes in the OS?
  - How do we pack up the process and set it aside?
  - How do we get a stack and heap for the kernel?
  - What are possible execution states, and how does the system move from one to another
  - …



A. Elkady

The process model

- Example: Multiprogramming of four programs
- **Conceptual model:**
  - 4 independent processes
  - Processes run sequentially
- **Reality:**
  - Only one program active at any instant!
  - That instant can be very short…
  - Only applies if there's a single CPU in the system



A. Elkady

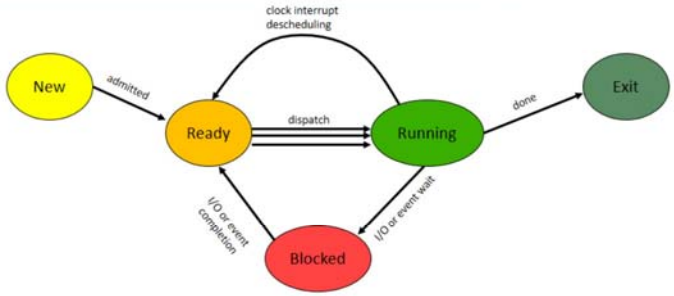
Process State

- There may be several processes running the same program *(e.g. multiple web browsers)*, but each is a distinct process with its own representation.
- Each process has an execution state that indicates what it is currently doing, e.g.,:
  - ready: waiting to be assigned to the CPU
  - running: executing instructions on the CPU
  - waiting: waiting for an event, e.g., I/O completion
- As a program executes, it moves from state to state

20

A. Elkady

Process State Transitions



- As a process executes, it changes state:
  - new: The process is being created
  - ready: The process is waiting to run
  - running: Instructions are being executed
  - blocked: Process waiting for some event to occur
  - terminated: The process has finished execution

A. Elkady

Change of Process State

- If the currently running process is to be moved to another state (Ready, Waiting, etc.), then the OS must make substantial changes in its environment
- Context Switch
  - Process of switching CPU from one process to another
  - Very machine dependent for types of registers

A. Elkady

PCB

- PCBs are data structures, dynamically allocated in OS memory.
- When a process is created, a PCB is allocated to it, initialized, and placed on the correct state queue (next).
- The PCB is where the OS keeps all of a process' hardware execution state (PC, SP, registers) when the process is not running.
- The current state of process held in a process control block (PCB):
  - This is a "snapshot" of the execution and protection environment contains all of the info about a process.
- As the process computes, its PCB moves from queue to queue.
- When the process is terminated, its PCB is deallocated.

PCB
Process state
Process number
Program counter
Stack pointer
Registers
Memory management info
Username of owner
List of open files
Scheduling information
Accounting info

A. Elkady 25

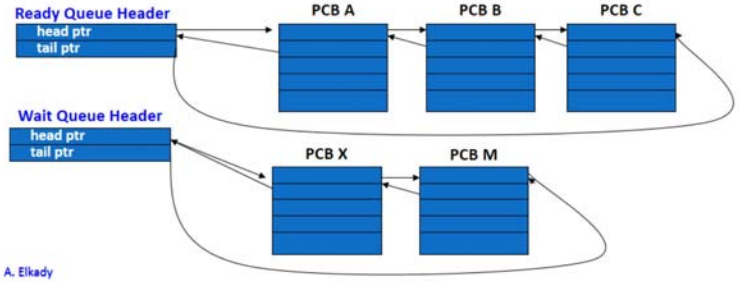
Details of Context Switching

- Very tricky to implement
  - OS must save state without changing state
  - Should run without touching any registers
    - CISC: single instruction saves all state
    - RISC: reserve registers for kernel
      - Or way to save a register and then continue
- Overheads: CPU is idle during a context switch
  - Explicit:
    - direct cost of loading/storing registers to/from main memory
  - Implicit:
    - Opportunity cost of flushing useful caches (cache, TLB, etc.)
    - Wait for pipeline to drain in pipelined processors

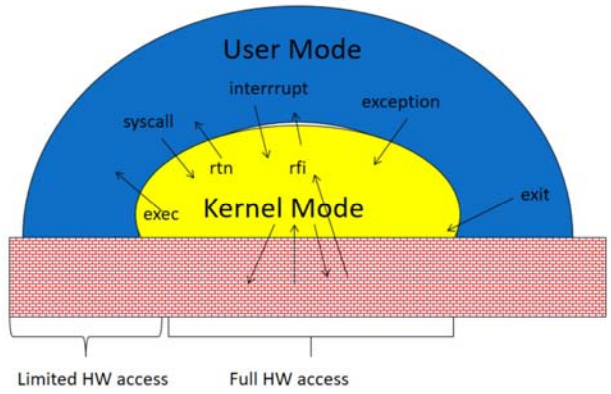
A. Elkady

State Queues

- The OS maintains a collection of queues that represent the state of all processes in the system
- There is typically **one queue for each state**, e.g., ready, waiting for I/O, etc.
- Each PCB is queued onto a state queue according to its current state.
  - As a process changes state, its PCB is unlinked from one queue and linked onto another.
- There may be many wait queues, one for eachtype of wait (specific device, timer, message,...).



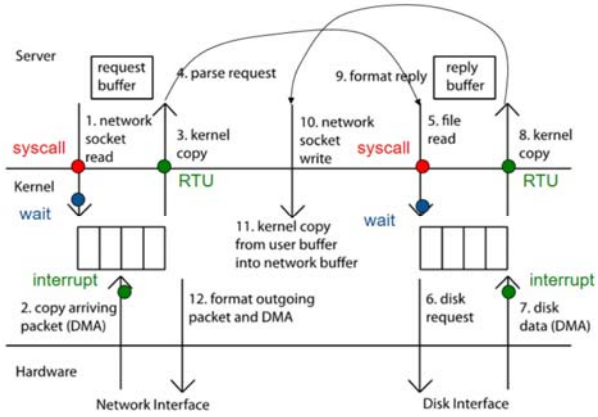
User/Kernal(Priviledged) Mode



4 OS concepts working together

- Privilege/User Mode
  - The hardware can operate in two modes, with only the "system" mode having the ability to access certain resources.
- Address Space
  - Programs execute in an address space that is distinct from the memory space of the physical machine
- Process
  - An instance of an executing program is a process consisting of an address space and one or more threads of control
- Protection
  - The OS and the hardware are protected from user programs and user programs are isolated from one another by controlling the translation from program virtual addresses to machine physical addresses

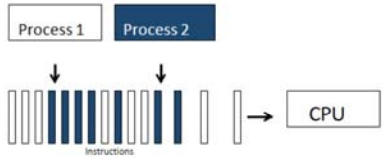
Putting it together: web server



Threads

Multitasking/Multithreading

- Even on systems with a single processing unit we may give the illusion of that several programs run at once

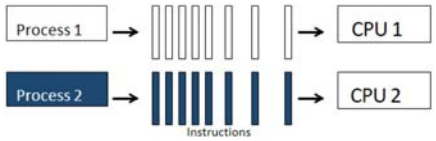


- Multitasking/multithreading: the operating system switches between the execution of different tasks/threads
- Execution of processes is said to be interleaved, as all are in progress, but only one is running at a time

A. Elkady

Remember: Multiprocessing

- Many of today's computations can take advantage of multiple processing units (multi-core processors)



- Multiprocessing: the use of more than one processing unit in a system
- Execution of processes is said to be parallel, as they are running at the same time

A. Elkady 39

Concurrency ≠ Parallelism

- Both multiprocessing and multitasking are examples of concurrent computation
- The execution of processes is said to be concurrent if it is either parallel or interleaved
- In this terminology, parallelism is a form of concurrency

In programming, the terms are often used to emphasize the type of problem they solve

- Concurrent programming: nondeterministic composition of independently executing processes
- Parallel programming: efficient execution of a deterministic computation on multiple processing units

A. Elkady 41

Processes and Threads

- A full process includes numerous things:
  - an address space (defining all the code and data pages)
  - OS resources and accounting information
  - a "thread of control", which defines where the process is currently executing (basically, the PC and registers)
- Creating a new process is costly, because of all of the structures (e.g., page tables) that must be allocated
- Communicating between processes is costly, because most communication goes through the OS

"Lightweight" Processes

- What's shared between these processes?
  - They all share the same code and data (address space)
  - they all share the same privileges
  - they share almost everything in the process
- What don't they share?
  - Each has its own PC, registers, and stack pointer
- Idea: why don't we separate the idea of process (address space, accounting, etc.) from that of the minimal "thread of control" (PC, SP, registers)?

Parallel Programs

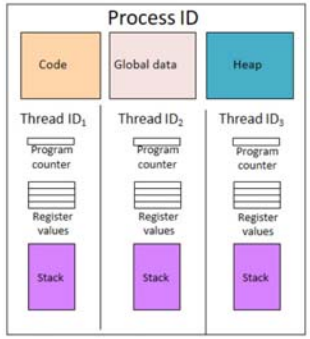
- Suppose I want to build a parallel program to execute on a multiprocessor, or a web server to handle multiple simultaneous web requests. I need to:
  - create several processes that can execute in parallel
  - cause each to map to the same address space (because they're part of the same computation)
  - give each its starting address and initial parameters
  - the OS will then schedule these processes, in parallel, on the various processors
- Notice that there's a lot of cost in creating these processes and possibly coordinating them. There's also a lot of duplication, because they all share the same address space, protection, etc.....

Threads and Processes

- Modern operating systems therefore support two entities:
  - the process, which defines the address space and general process attributes
  - the thread, which defines a sequential execution stream within a process
- A thread is bound to a single process. For each process, however, there may be many threads.
- Threads are the unit of scheduling; processes are containers in which threads execute.

Threads

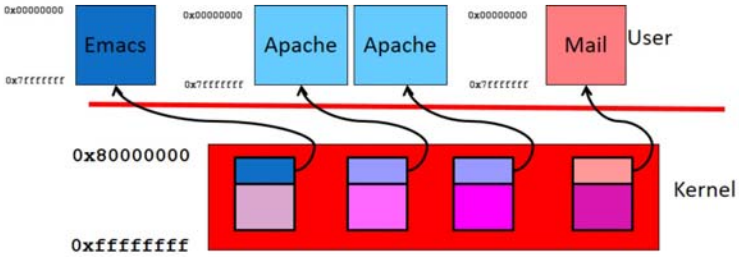
- Make programs concurrent by associating them with threads
- A thread is a part of an operating system process
- Private components
  - Thread identifier
  - Thread state
  - Thread context
  - Memory: only stack
- Shared components
  - Program text *(code)*
  - Global data
  - Heap



A. Elkady 21

Processes and Address Spaces

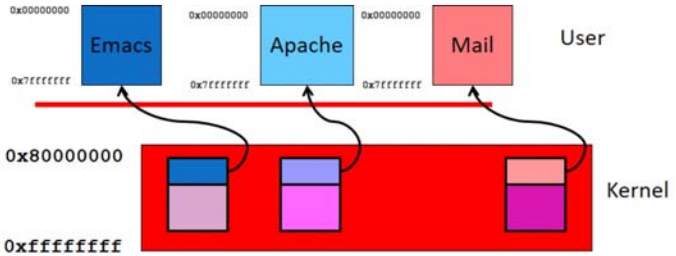
- Two heavyweight address spaces for two concurrent computations ?



A. Elkady 48

Processes and Address Spaces

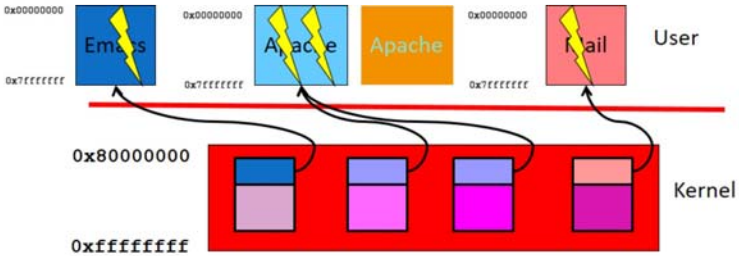
- What happens when Apache wants to run multiple concurrent computations ?



A. Elkady 47

Processes and Address Spaces

- We can eliminate duplicate address spaces and place concurrent computations in the same address space



A. Elkady 49

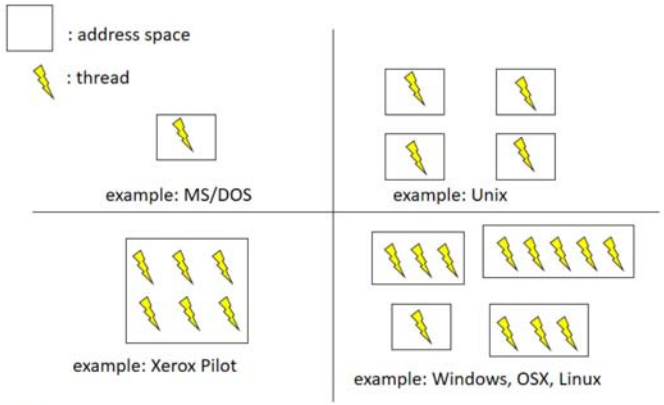
Threads

- Lighter weight than processes
- Threads need to be mutually trusting
  - Why?
- Ideal for programs that want to support concurrent computations where lots of code and data are shared between computations
  - Servers, GUI code, …

Separation of Threads and Processes

- Separating threads and processes makes it easier to support multi-threaded applications
- Concurrency (multi-threading) is useful for:
  - improving program structure
  - handling concurrent events (e.g., web requests)
  - building parallel programs
- So, multi-threading is useful even on a uniprocessor
- To be useful, thread operations have to be fast

How different OSes support threads

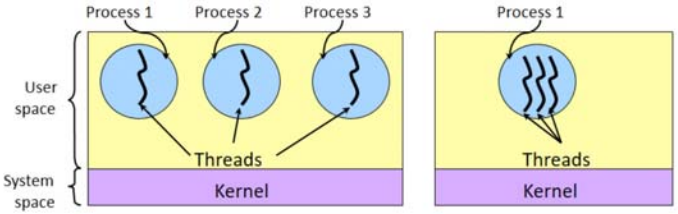


Modern Process with Threads

- Thread: a sequential execution stream within process (Sometimes called a "Lightweight process")
  - Process still contains a single Address Space
  - No protection between threads
- Multithreading: a single program made up of a number of different concurrent activities
  - Sometimes called multitasking, as in Ada …
- Why separate the concept of a thread from that of a process?
  - Access the "thread" part of a process (concurrency)
  - Separate from the "address space" (protection)
  - Heavyweight Process ≡ Process with one thread

Threads: "processes" sharing memory

- Thread: a sequential execution stream within process (Sometimes called a "Lightweight process")
  - Process still contains a single Address Space
  - No protection between threads
- **Process** == address space
- **Thread** == program counter / stream of instructions



A. Elkady

Processes vs. Threads

- Creating and managing processes is generally regarded as an expensive task (fork system call).
- Making sure all the processes peacefully co-exist on the system is not easy (as concurrency transparency comes at a price).
- Threads can be thought of as an "execution of a part of a program (in user-space)".
- Rather than make the OS responsible for concurrency transparency, it is left to the individual application to manage the creation and scheduling of each thread.

A. Elkady

Threads Motivation

- Most modern applications are multithreaded.
- Threads run within application.
- Multiple tasks with the application can be implemented by separate threads:
  - Update display
  - Fetch data
  - Spell checking
  - Answer a network request
- Process creation is heavy-weight while thread creation is light-weight.
- Can simplify code, increase efficiency.
- Kernels are generally multithreaded.

A. Elkady

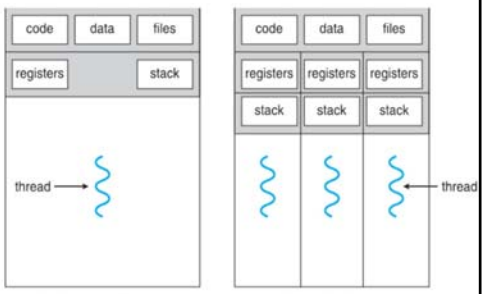
Thread Characteristics

- Has an execution context/state.
- Thread context is saved when not running.
- Has an execution stack and some per-thread static storage for local variables.
- Has access to the memory address space and resources of its task:
  - all threads of a task share this.
  - when one thread alters a (non-private) memory item, all other threads (of the task) see that.
  - a file open with one thread, is available to others.

A. Elkady

Single and Multithreaded Processes

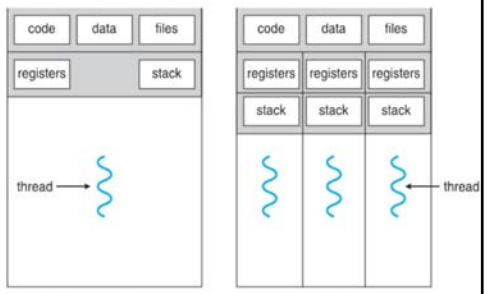
- Why have multiple threads per address space?
  - Sometimes need parallelism for a single job, and processes are very expensive – to start, switch between, and to communicate between
- Suspending a process involves suspending all threads of the process since all threads share the same address space
- Termination of a process, terminates all threads within the process



A. Elkady

Why Use Threads?

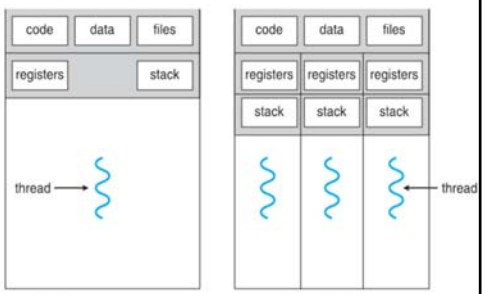
- Utilize multiple CPU's concurrently
- Low cost communication via shared memory
- Overlap computation and blocking on a single CPU
  - Blocking due to I/O
  - Computation and communication
- Handle asynchronous events



A. Elkady

Concurrent Access to Shared State

- Important: Changes made to shared state by one thread will be visible to the others!
- Reading and writing memory locations requires synchronization!
- This is a major topic for later …



A. Elkady

Benefits of Threads (1)

- Responsiveness – may allow continued execution if part of process is blocked, especially important for user interfaces.
- Resource Sharing – threads share process resources, easier than shared memory or message passing.
- Economy – cheaper than process creation, thread switching lower overhead than context switching.
- Scalability – process can take advantage of multiprocessor architectures and networked/distributed systems.

A. Elkady

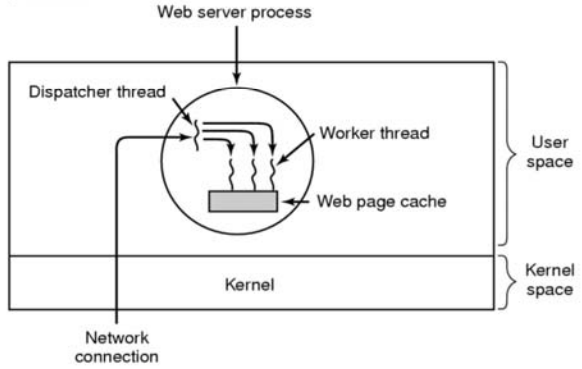
Benefits of Threads (2)

- Threads allows parallel activity inside a single address space:
  - While one server thread is blocked and waiting, a second thread in the same task can run.
  - Less time to create and terminate a thread than a process (because we do not need another address space).
  - Less time to switch between two threads than between processes.
  - Inter-thread communication and synchronization is very fast.

A. Elkady

Examples of benefits of threads

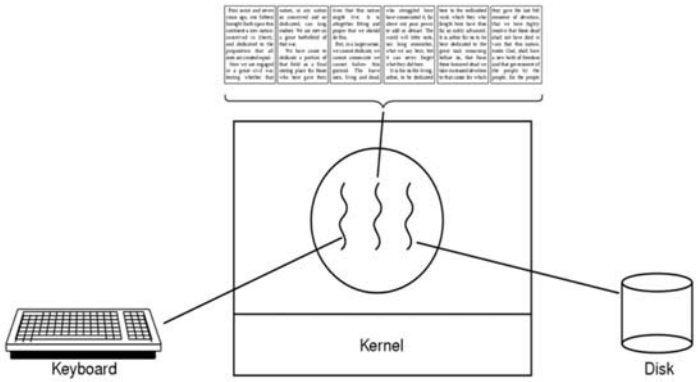
- Example 2: a File/Web server on a LAN:
  - It needs to handle several files/pages requests over a short period.
  - Hence more efficient to create (and destroy) a single thread for each request.
  - On a SMP machine: multiple threads can possibly be executing simultaneously on different processors.



A. Elkady

Examples of benefits of threads

- Example 1: Word Processor:
  - one thread displays menu and reads user input while another thread executes user commands and a third one writes to disk – the application is more responsive.



A. Elkady

Web Server Threads

```
while (TRUE) {
  get_next_request(&buf);
  handoff_work(&buf);
}
```

(a)
Dispatcher thread

```
while (TRUE) {
  wait_for_work(&buf)
  look_for_page_in_cache(&buf, &page);
  if (page_not_in_cache(&page)
      read_page_from_disk(&buf, &page);
  return_page(&page);
}
```

(b)
Worker thread

A. Elkady

Important Implications

- Two Important Implications:
  - Threaded applications often run faster than non-threaded applications (as context-switches between kernel and user-space are avoided).
  - Threaded applications are harder to develop (although simple, clean designs can help here).
- Additionally, the assumption is that the development environment provides a Threads Library for developers to use (most modern environments do).

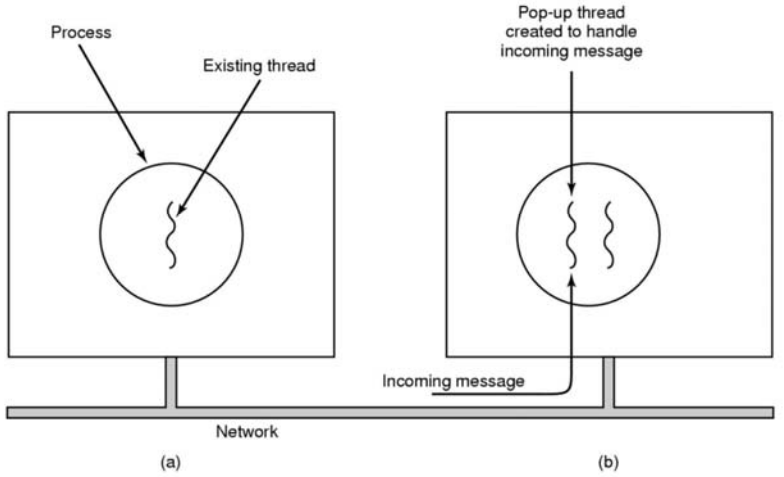
Application benefits of threads (2)

- Since threads within the same process share memory and files, they can communicate with each other without invoking the kernel.
- Therefore necessary to synchronize the activities of various threads so that they do not obtain inconsistent views of the current data.

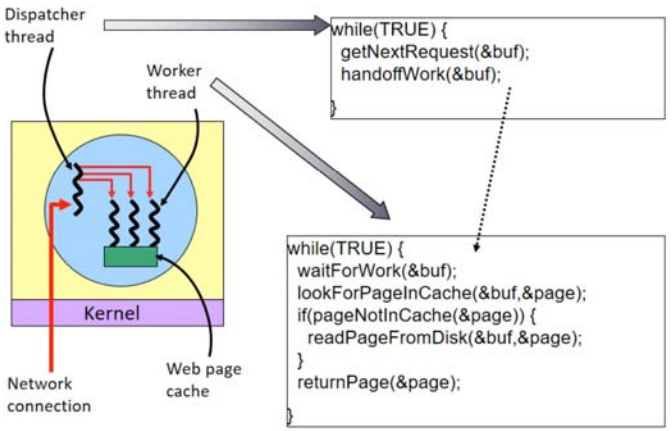
Application benefits of threads (1)

- Consider an application that consists of several independent parts that do not need to run in sequence.
- Each part can be implemented as a thread.
- Whenever one thread is blocked waiting for an I/O, execution could possibly switch to another thread of the same application (instead of blocking it and switching to another application).

Pop-Up / Worker Threads



Multithreaded Web server



User-Level Threads

- To make threads really fast, they should be implemented at the user level
- A user-level thread is managed entirely by the run-time system (user-level code that is linked with your program).
- Each thread is represented simply by a PC, registers, stack and a little control block, managed in the user's address space.
- Creating a new thread, switching between threads, and synchronizing between threads can all be done without kernel involvement

Kernel Threads

- Kernel threads still suffer from performance problems
- Operations on kernel threads are slow because:
  - a thread operation still requires a kernel call
  - kernel threads may be overly general, in order to support needs of different users, languages, etc.
  - the kernel doesn't trust the user, so there must be lots of checking on kernel calls

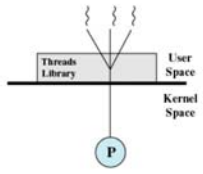
Implementing Threads

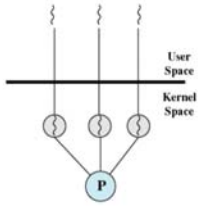
USER-LEVEL THREADS

- All thread management is done by the application
- The kernel is not aware of the existence of threads

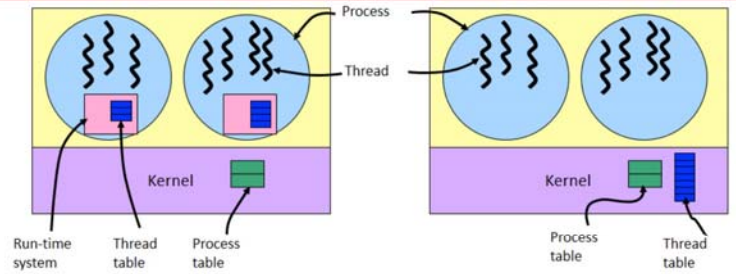
KERNEL-LEVEL THREADS

- Windows is an example of this approach
- Kernel maintains context information for the process and the threads
- Scheduling is done on a thread basis





User-Level Vs. Kernel-Leve Threads



User-level threads

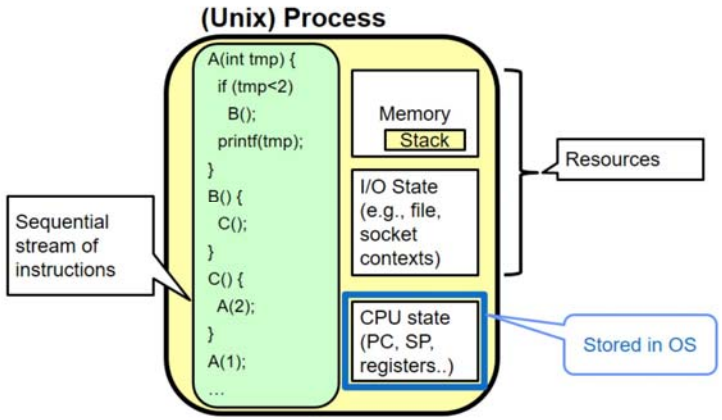
+ No need for kernel support
- May be slower than kernel threads
- Harder to do non-blocking I/O
- Process as whole gets one time slice irrespective of whether process has one thread or 1000 threads

Kernel-level threads

+ More flexible scheduling
+ Non-blocking I/O
- Not portable
+ Each thread gets a scheduling slice

A. Elkady 83

Putting it together: Process



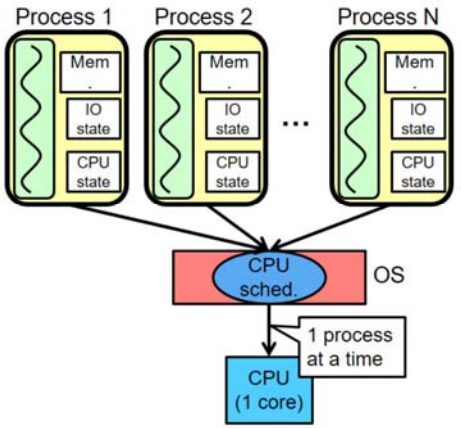
A. Elkady

Examples multithreaded programs

- Embedded systems
  - Elevators, Planes, Medical systems, Wristwatches
  - Single Program, concurrent operations
- Most modern OS kernels
  - Have to deal with concurrent requests by multiple users
  - But no protection needed within kernel
- Database Servers
  - Access to shared data by many concurrent users
  - Also background utility processing must be done
- Network Servers
  - Concurrent requests from network
  - Again, single program, multiple concurrent operations
  - File server, Web server, and airline reservation systems
- Parallel Programming (More than one physical CPU)
  - Split program into multiple threads for parallelism
  - This is called Multiprocessing

A. Elkady

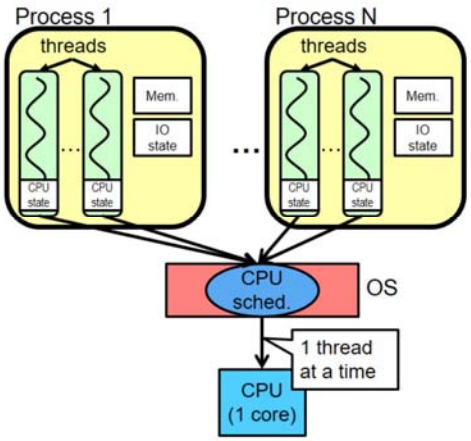
Putting it together: Processes



- Switch overhead: high
  - CPU state: low
  - Memory/IO state: high
- Process creation: high
- Protection
  - CPU: yes
  - Memory/IO: yes
- Sharing overhead: high (involves at least a context switch)

A. Elkady

Putting it together: Threads

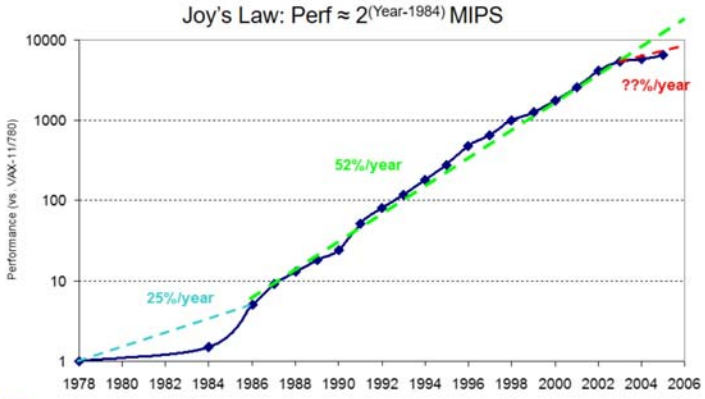


- Switch overhead: low (only CPU state)
- Thread creation: low
- Protection
  - CPU: yes
  - Memory/IO: No
- Sharing overhead: low (thread switch overhead low)

A. Elkady

Challenge: Slowdown in Joy's law of Performance

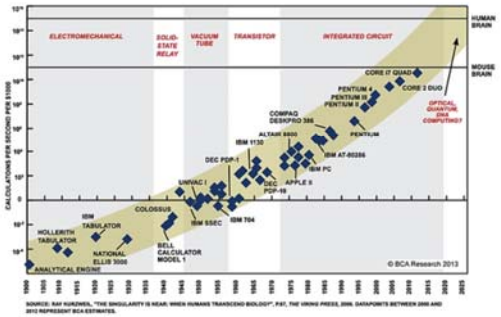
- Joy's Law, first formulated by Sun Microsystems co-founder Bill Joy in 1983, states that the peak computer speed doubles each year and thus is given by a simple function of time.



A. Elkady 93

Technology Trends: Moore's Law

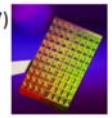
Gordon Moore (co-founder of Intel) predicted in 1965 that the **transistor density** of semiconductor chips would **double roughly every 18 months**.



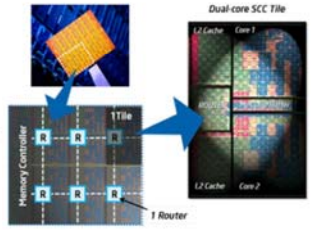
A. Elkady

ManyCore Chips: The future is here

- Intel 80-core multicore chip (Feb 2007)
  - 80 simple cores
  - Two FP-engines / core
  - Mesh-like network
  - 100 million transistors



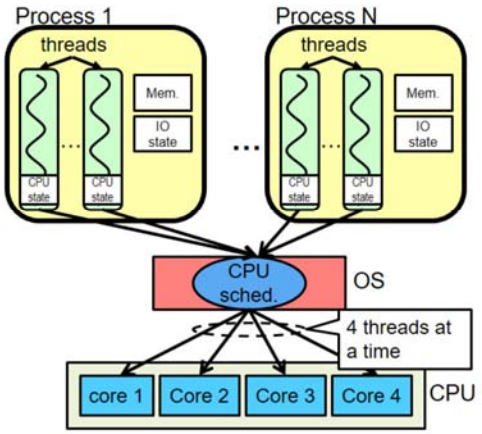
- Intel Single-Chip Cloud Computer (August 2010)
  - 24 "tiles" with two cores/tile
  - 24-router mesh network
  - 4 DDR3 memory controllers
  - Hardware support for message-passing



- "ManyCore" refers to many processors/chip
  - 64? 128? Hard to say exact boundary
- How to program these?
  - Use 2 CPUs for video/audio
  - Use 1 for word processor, 1 for browser
  - 76 for virus checking???
- Parallelism must be exploited at all levels

A. Elkady

Putting it together: Multi-Cores

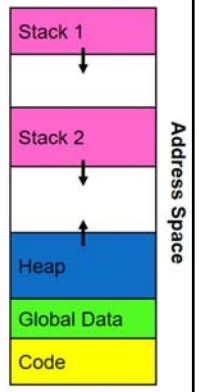


- Switch overhead: low (only CPU state)
- Thread creation: low
- Protection
  - CPU: yes
  - Memory/IO: No
- Sharing overhead: low (thread switch overhead low)

A. Elkady

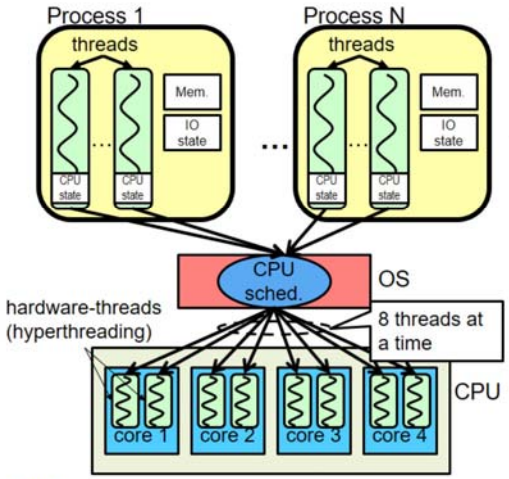
Memory Footprint: Two-Threads

- If we stopped this program and examined it with a debugger, we would see
  - Two sets of CPU registers
  - Two sets of Stacks
- Questions:
  - How do we position stacks relative to each other?
  - What maximum size should we choose for the stacks?
  - What happens if threads violate this?
  - How might you catch violations?



A. Elkady

Putting it together: Hyper-Threading

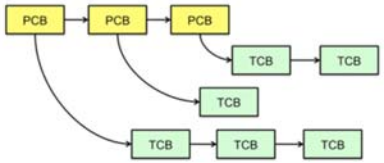


- Switch overhead between hardware-threads: very-low (done in hardware)
- Cache-resource contention by ALUs/FPUs may hurt performance
  - If an application has been optimized for a specific cache size and number of physical processors
  - As a result: each logical processor would try to fully utilize the cache

A. Elkady

Multithreaded Processes

- PCB points to multiple TCBs:



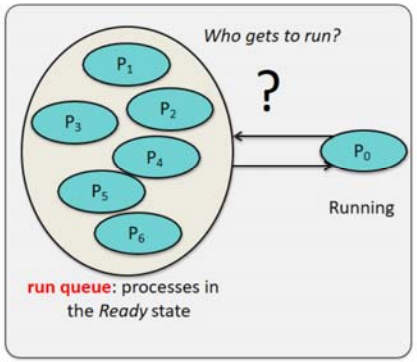
- Switching threads within a block is a simple thread switch
- Switching threads across blocks requires changes to memory and I/O address tables.

A. Elkady

Scheduling

Scheduling

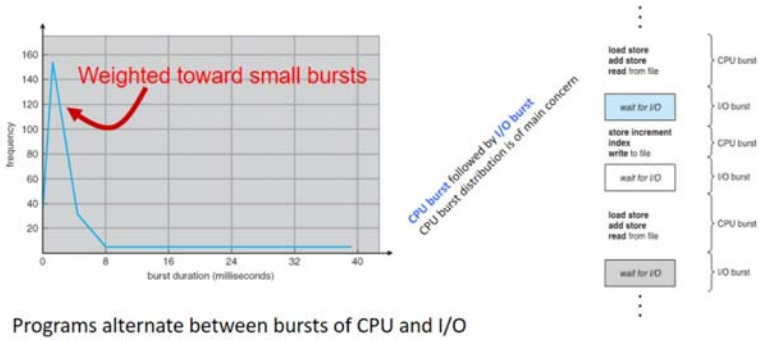
- We have multiple tasks *ready* to run, which one should we choose?
- Scheduling algorithm:
  - = Policy
  - Makes the decision of who gets to run
- Dispatcher:
  - = Mechanism
  - To do the context switch



Scheduling

- the art, theory, and practice of deciding what to do next
- Ex: FIFO non-premptive scheduling
- Ex: Round-Robin
- Ex: Priority-based

CPU Bursts



Programs alternate between bursts of CPU and I/O

- Program typically uses the CPU for some period of time, then does I/O, then uses CPU again
- Each scheduling decision is about which job to give to the CPU for use by its next CPU burst
- With preemption, thread may be forced to give up CPU before finishing current CPU burst

How to Handle Simultaneous Mix of Diff Types of Apps?

- Can we use Burst Time (observed) to decide which application gets CPU time?
- Consider mix of interactive and high throughput apps:
  - How to best schedule them?
  - How to recognize one from the other?
    - Do you trust app to say that it is "interactive"?
  - Should you schedule the set of apps identically on servers, workstations, pads, and cellphones?
- Assumptions encoded into many schedulers:
  - Apps that sleep a lot and have short bursts must be interactive apps – they should get high priority
  - Apps that compute a lot should get low(er?) priority, since they won't notice intermittent bursts from interactive apps
- Hard to characterize apps:
  - What about apps that sleep for a long time, but then compute for a long time?
  - Or, what about apps that must run under all circumstances (say periodically)

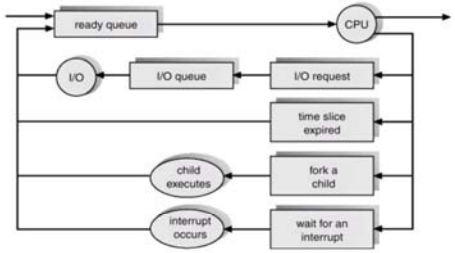
A. Elkady

Measuring scheduling performance

- In order to achieve an efficient processor management, OS tries to select the most appropriate process from the ready queue.
- For selection, the relative importance of the followings may be considered as performance criteria.

A. Elkady

Processor Scheduling



- life-cycle of a thread
  - Active threads work their way from Ready queue to Running to various waiting queues.
- Scheduling: deciding which threads are given access to resources
- How to decide which of several threads to dequeue and run?
  - So far we have a single ready queue
  - Reason for wait->ready may make a big difference!

A. Elkady

Performance Criteria

- Processor Utilization:
  - The ratio of busy time of the processor to the total time to finish all processes.
  - Processor Utilization = (Processor busy time) / (Processor busy time + Processor idle time)
  - We would like to keep the processor as busy as possible.
- Throughput:
  - The measure of work done in a unit time interval.
  - Throughput = (Number of processes completed) / (Time Unit)

We normally want to *Maximize* these

A. Elkady 123

Performance Criteria

- Turnaround Time (tat):
  - The sum of time spent in the ready queue, execution time and I/O time.
  - tat = t(process completed) – t(process submitted)
- Waiting Time (wt):
  - Total time spent only in ready queue.
  - Processor scheduling algorithms only affect the time spent waiting in the ready queue. So, considering only waiting time instead of turnaround time is generally sufficient.
- Response Time (rt):
  - The amount of time it takes to start responding. This criterion is important for interactive systems.
  - rt = t(first response) – t(submission of request)

Scheduling Policy Goals/Criteria

- Minimize Response Time
  - Minimize elapsed time to do an operation (or job)
  - Response time is what the user sees:
    - Time to echo a keystroke in editor
    - Time to compile a program
    - Real-time Tasks: Must meet deadlines imposed by World
- Maximize Throughput
  - Maximize operations (or jobs) per second
  - Throughput related to response time, but not identical:
    - Minimizing response time will lead to more context switching than if you only maximized throughput
  - Two parts to maximizing throughput
    - Minimize overhead (for example, context-switching)
    - Efficient use of resources (CPU, disk, memory, etc)
- Fairness
  - Share CPU among users in some equitable way
  - Fairness is not minimizing average response time:
    - Better average response time by making system less fair

Performance Criteria

- We, normally, want to:
  - Maximize the processor utilization and throughput, and
  - Minimize *tat, wt,* and *rt*.
- However, sometimes other combinations may be considered depending on the system requirements.
- For example, for the time sharing systems, response time is quite important.

How To Give Up The CPU?

Schedulers can operate with or without cooperation:

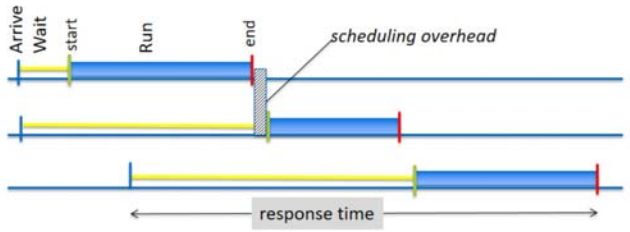
- Voluntary yield (cooperative multitasking) – once a process is in the running state, it will continue until it terminates or blocks itself for I/O
  - Relies on "Scout's Honor" programming
    - Assumes that everyone plays nicely
    - If one process doesn't, it can block everything else!
  - Example: older versions of MacOS, Windows
- Involuntary yield (Preemption) – running process may have CPU taken away from it forcibly and moved to ready state by the OS
  - Preemption may occur when new process arrives, or an interrupt, or periodically
  - Example: most multitasking systems (UNIX†, NT, VMS, etc.)

When Does The Scheduler Make Decisions?

- Four events may cause the scheduler to get called:
  1. Current process goes from *running* to *blocked* state
  2. Current process terminates
  3. Interrupt gives the scheduler a chance to move a process from *running* to *ready*: *scheduler decides it's time for someone else to run*
  4. Current process goes from *blocked* to *ready*
     I/O is complete (including blocking events, such as semaphores)
     This does not necessarily mean the currently running process will change
- Preemptive scheduler
- Cooperative (non-preemptive) scheduler
- – CPU cannot be taken away unless a system call takes place or process exits
- Run-to-completion scheduler (old batch systems)

Processor Scheduling algorithms

- The goal of a scheduling algorithm is to select the most appropriate process in the ready queue.
- For simplicity, we will assume that
  - we have a single I/O server and a single device queue,
  - device queue always implemented with FCFS (FIFO) method.
- We also will neglect the context switching overhead between processes.



Interactive vs. batch scheduling

Batch

- First-Come-First-Served (FCFS) *(non-preemptive)*
- Shortest Job/Process First (SJF/SPF) *(non-preemptive)*
- Shortest Remaining Time First (SRTF) *(preemptive)*
- Priority *(preemptive /non-preemptive)*

Interactive

- Round-Robin (RR) Priority *(preemptive)*
- Lottery

Processor Scheduling algorithms

- We will consider the following processes and will apply different scheduling algorithms on it.

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4	4	4	4	4
B	2	8	1	8	-	-
C	3	2	1	2	-	-
D	7	1	1	1	1	1

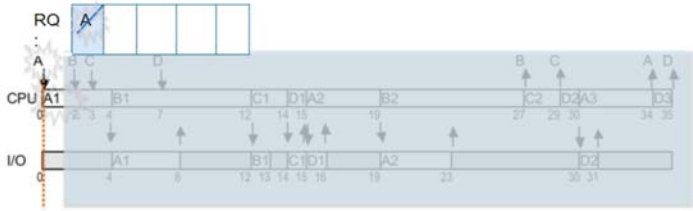
First-Come-First-Served (FCFS)

- In this algorithm, the process to be selected is the process which requests the processor first.
- This is the process whose PCB is at the head of the ready queue.
- Contrary to its simplicity, its performance may often be poor compared to other algorithms.

A. Elkady

First-Come-First-Served (FCFS)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4	4	4	4	4
B	2	8	1	8	-	-
C	3	2	1	2	-	-
D	7	1	1	1	1	1



A. Elkady

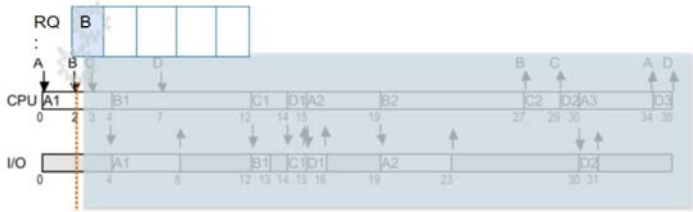
First-Come-First-Served (FCFS)

- FCFS may cause processes with short processor bursts to wait for a long time.
- If one process with a long processor burst gets the processor, all the others will wait for it to release it and the ready queue will be crowded.
- This is called the convoy *effect*.

A. Elkady

First-Come-First-Served (FCFS)

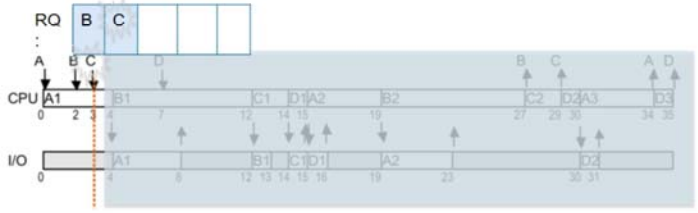
Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4	4	4	4	4
B	2	8	1	8	-	-
C	3	2	1	2	-	-
D	7	1	1	1	1	1



A. Elkady

First-Come-First-Served (FCFS)

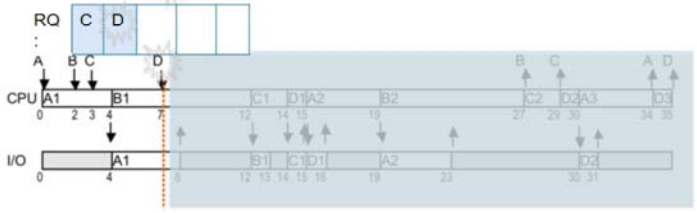
Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4	4	4	4	4
B	2	8	1	8	-	-
C	3	2	1	2	-	-
D	7	1	1	1	1	1



A. Elkady

First-Come-First-Served (FCFS)

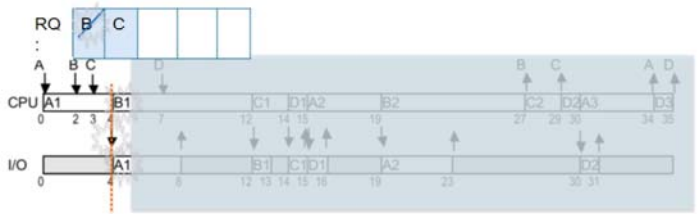
Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4	4	4	4	4
B	2	8	1	8	-	-
C	3	2	1	2	-	-
D	7	1	1	1	1	1



A. Elkady

First-Come-First-Served (FCFS)

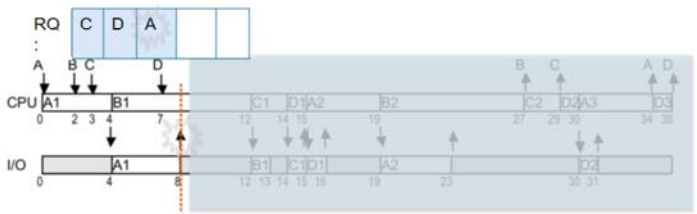
Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4	4	4	4	4
B	2	8	1	8	-	-
C	3	2	1	2	-	-
D	7	1	1	1	1	1



A. Elkady

First-Come-First-Served (FCFS)

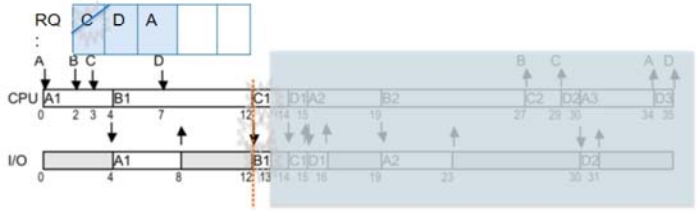
Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4	4	4	4	4
B	2	8	1	8	-	-
C	3	2	1	2	-	-
D	7	1	1	1	1	1



A. Elkady

2.3.1 First-Come-First-Served (FCFS)

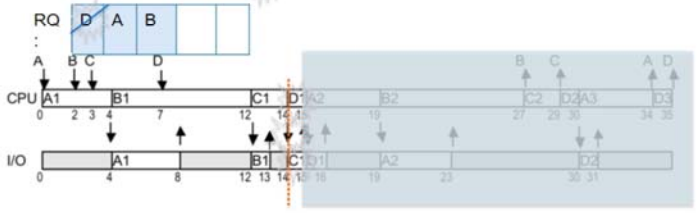
Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4	4	4	4	4
B	2	8	1	8	-	-
C	3	2	1	2	-	-
D	7	1	1	1	1	1



A. Elkady

First-Come-First-Served (FCFS)

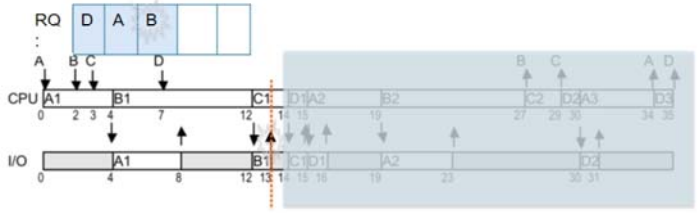
Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4	4	4	4	4
B	2	8	1	8	-	-
C	3	2	1	2	-	-
D	7	1	1	1	1	1



A. Elkady

First-Come-First-Served (FCFS)

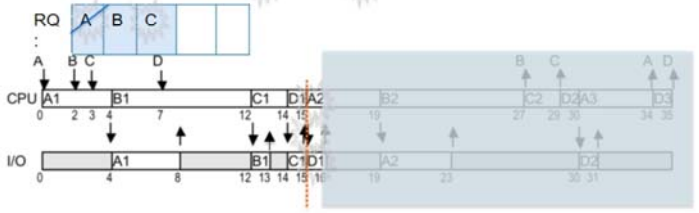
Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4	4	4	4	4
B	2	8	1	8	-	-
C	3	2	1	2	-	-
D	7	1	1	1	1	1



A. Elkady

First-Come-First-Served (FCFS)

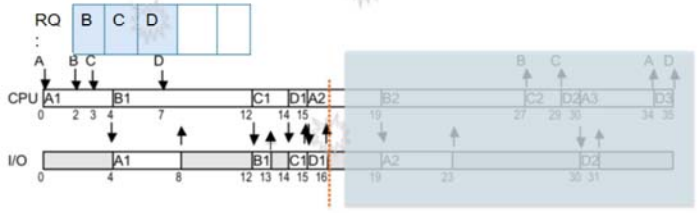
Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4	4	4	4	4
B	2	8	1	8	-	-
C	3	2	1	2	-	-
D	7	1	1	1	1	1



A. Elkady

First-Come-First-Served (FCFS)

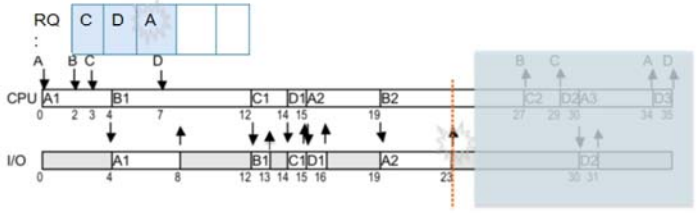
Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4	4	4	4	4
B	2	8	1	8	-	-
C	3	2	1	2	-	-
D	7	1	1	1	1	1



A. Elkady

First-Come-First-Served (FCFS)

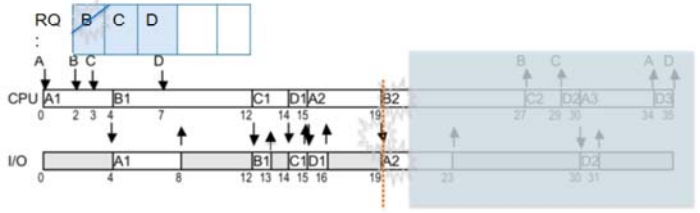
Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4	4	4	4	4
B	2	8	1	8	-	-
C	3	2	1	2	-	-
D	7	1	1	1	1	1



A. Elkady

First-Come-First-Served (FCFS)

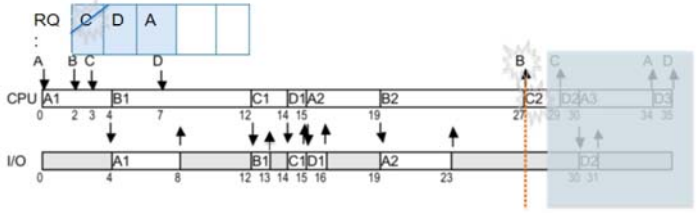
Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4	4	4	4	4
B	2	8	1	8	-	-
C	3	2	1	2	-	-
D	7	1	1	1	1	1



A. Elkady

First-Come-First-Served (FCFS)

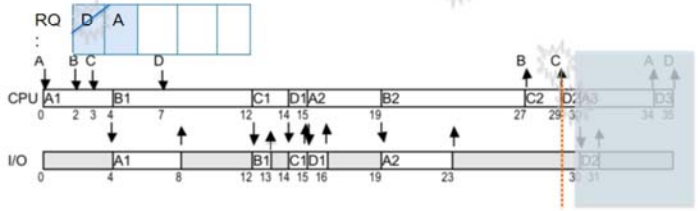
Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4	4	4	4	4
B	2	8	1	8	-	-
C	3	2	1	2	-	-
D	7	1	1	1	1	1



A. Elkady

First-Come-First-Served (FCFS)

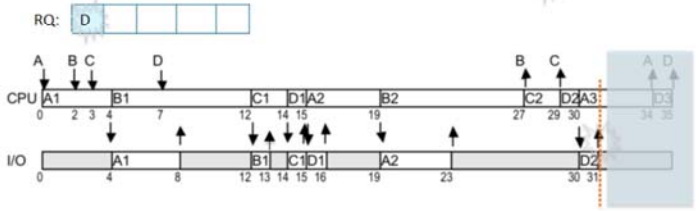
Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4	4	4	4	4
B	2	8	1	8	-	-
C	3	2	1	2	-	-
D	7	1	1	1	1	1



A. Elkady

First-Come-First-Served (FCFS)

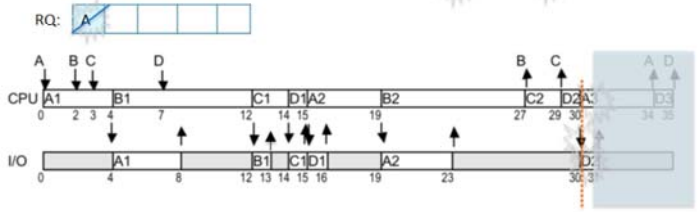
Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4	4	4	4	4
B	2	8	1	8	-	-
C	3	2	1	2	-	-
D	7	1	1	1	1	1



A. Elkady

First-Come-First-Served (FCFS)

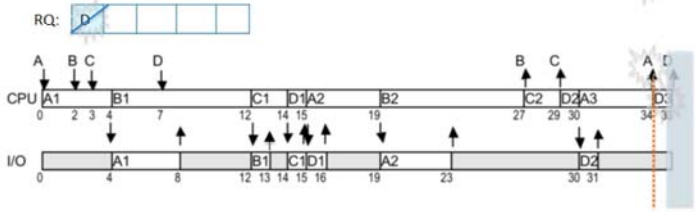
Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4	4	4	4	4
B	2	8	1	8	-	-
C	3	2	1	2	-	-
D	7	1	1	1	1	1



A. Elkady

First-Come-First-Served (FCFS)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4	4	4	4	4
B	2	8	1	8	-	-
C	3	2	1	2	-	-
D	7	1	1	1	1	1

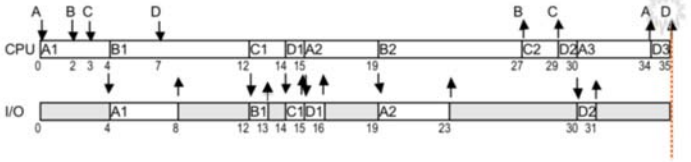


A. Elkady

First-Come-First-Served (FCFS)

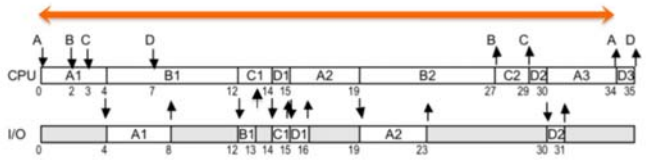
Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4	4	4	4	4
B	2	8	1	8	-	-
C	3	2	1	2	-	-
D	7	1	1	1	1	1

RQ:



A. Elkady

First-Come-First-Served (FCFS)

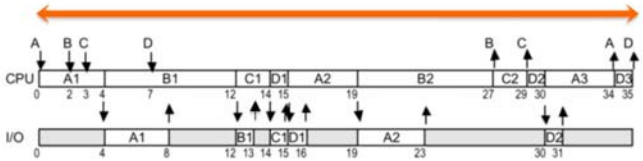


- Turn around time:
  - $tat_A = 34 - 0 = 34$
  - $tat_B = 27 - 2 = 25$
  - $tat_C = 29 - 3 = 26$
  - $tat_D = 35 - 7 = 28$

$tat_{AVG} = (34 + 25 + 26 + 28) / 4 = 28.25$

A. Elkady

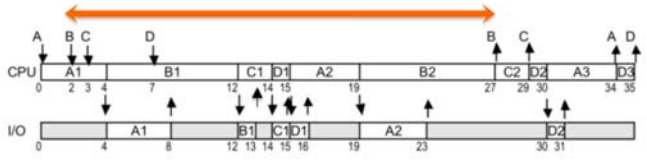
First-Come-First-Served (FCFS)



- Processor utilization = (35 / 35) * 100 = 100 %
- Throughput = 4 / 35=0.11

A. Elkady

First-Come-First-Served (FCFS)

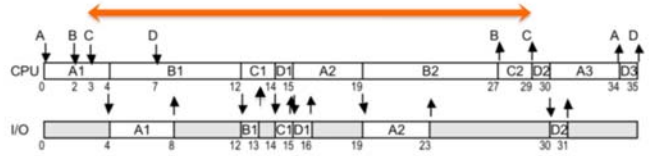


- Turn around time:
  - $tat_A = 34 - 0 = 34$
  - $tat_B = 27 - 2 = 25$
  - $tat_C = 29 - 3 = 26$
  - $tat_D = 35 - 7 = 28$

$tat_{AVG} = (34 + 25 + 26 + 28) / 4 = 28.25$

A. Elkady

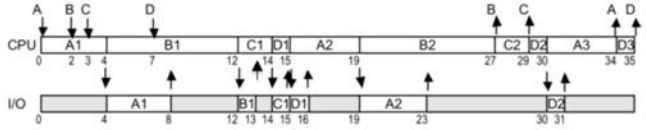
First-Come-First-Served (FCFS)



- Turn around time:
  - $tat_A = 34 - 0 = 34$
  - $tat_B = 27 - 2 = 25$
  - $tat_C = 29 - 3 = 26$
  - $tat_D = 35 - 7 = 28$

  $tat_{AVG} = (34 + 25 + 26 + 28) / 4 = 28.25$

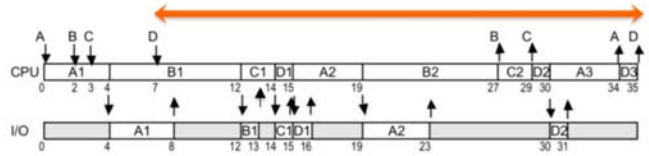
First-Come-First-Served (FCFS)



- Turn around time:
  - $tat_A = 34 - 0 = 34$
  - $tat_B = 27 - 2 = 25$
  - $tat_C = 29 - 3 = 26$
  - $tat_D = 35 - 7 = 28$

  $tat_{AVG} = (34 + 25 + 26 + 28) / 4 = 28.25$

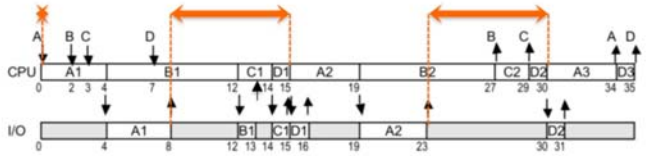
First-Come-First-Served (FCFS)



- Turn around time:
  - $tat_A = 34 - 0 = 34$
  - $tat_B = 27 - 2 = 25$
  - $tat_C = 29 - 3 = 26$
  - $tat_D = 35 - 7 = 28$

  $tat_{AVG} = (34 + 25 + 26 + 28) / 4 = 28.25$

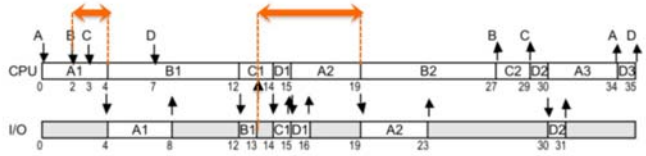
First-Come-First-Served (FCFS)



- Waiting time:
  - $wt_A = (0 - 0) + (15 - 8) + (30 - 23) = 14$
  - $wt_B = (4 - 2) + (19 - 13) = 8$
  - $wt_C = (12 - 3) + (27 - 15) = 21$
  - $wt_D = (14 - 7) + (29 - 16) + (34 - 31) = 23$

  $wt_{AVG} = (14 + 12 + 21 + 23) / 4 = 16.5$

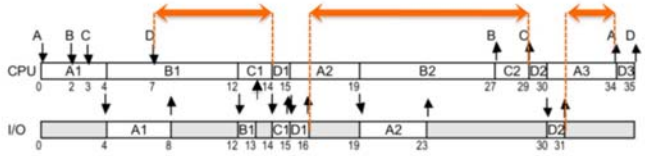
First-Come-First-Served (FCFS)



- Waiting time:
  - $wt_A = (0 - 0) + (15 - 8) + (30 - 23) = 14$
  - $wt_B = (4 - 2) + (19 - 13) = 8$
  - $wt_C = (12 - 3) + (27 - 15) = 21$
  - $wt_D = (14 - 7) + (29 - 16) + (34 - 31) = 23$
  - $wt_{AVG} = (14 + 12 + 21 + 23) / 4 = 16.5$

A. Elkady

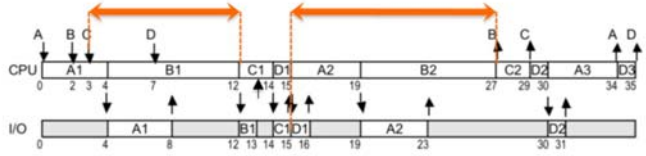
First-Come-First-Served (FCFS)



- Waiting time:
  - $wt_A = (0 - 0) + (15 - 8) + (30 - 23) = 14$
  - $wt_B = (4 - 2) + (19 - 13) = 8$
  - $wt_C = (12 - 3) + (27 - 15) = 21$
  - $wt_D = (14 - 7) + (29 - 16) + (34 - 31) = 23$
  - $wt_{AVG} = (14 + 12 + 21 + 23) / 4 = 16.5$

A. Elkady

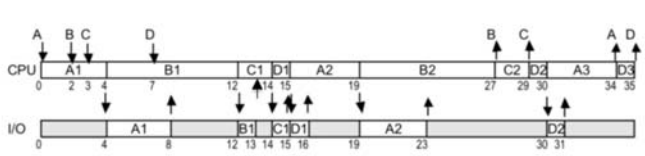
First-Come-First-Served (FCFS)



- Waiting time:
  - $wt_A = (0 - 0) + (15 - 8) + (30 - 23) = 14$
  - $wt_B = (4 - 2) + (19 - 13) = 8$
  - $wt_C = (12 - 3) + (27 - 15) = 21$
  - $wt_D = (14 - 7) + (29 - 16) + (34 - 31) = 23$
  - $wt_{AVG} = (14 + 12 + 21 + 23) / 4 = 16.5$

A. Elkady

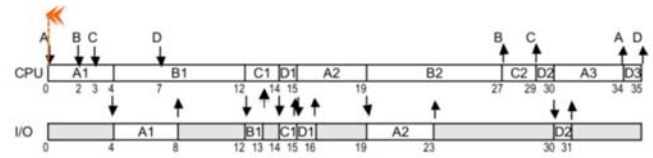
First-Come-First-Served (FCFS)



- Waiting time:
  - $wt_A = (0 - 0) + (15 - 8) + (30 - 23) = 14$
  - $wt_B = (4 - 2) + (19 - 13) = 8$
  - $wt_C = (12 - 3) + (27 - 15) = 21$
  - $wt_D = (14 - 7) + (29 - 16) + (34 - 31) = 23$
  - $wt_{AVG} = (14 + 12 + 21 + 23) / 4 = 16.5$

A. Elkady

First-Come-First-Served (FCFS)

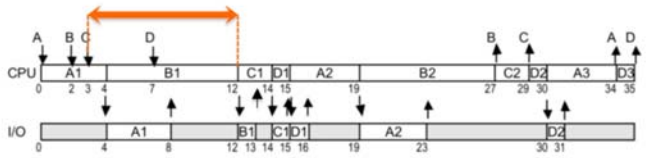


- Response time:
  - $rt_A = 0 - 0 = 0$
  - $rt_B = 4 - 2 = 2$
  - $rt_C = 12 - 3 = 9$
  - $rt_D = 14 - 7 = 7$

  $rt_{AVG} = (0 + 2 + 9 + 7) / 4 = 4.5$

A. Elkady

First-Come-First-Served (FCFS)

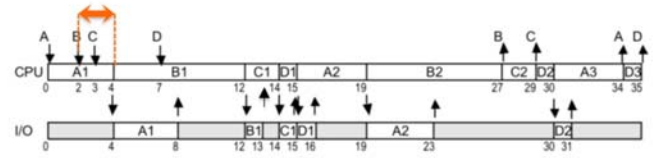


- Response time:
  - $rt_A = 0 - 0 = 0$
  - $rt_B = 4 - 2 = 2$
  - $rt_C = 12 - 3 = 9$
  - $rt_D = 14 - 7 = 7$

  $rt_{AVG} = (0 + 2 + 9 + 7) / 4 = 4.5$

A. Elkady

First-Come-First-Served (FCFS)

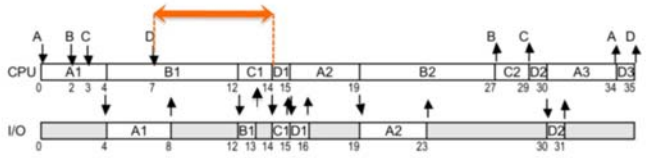


- Response time:
  - $rt_A = 0 - 0 = 0$
  - $rt_B = 4 - 2 = 2$
  - $rt_C = 12 - 3 = 9$
  - $rt_D = 14 - 7 = 7$

  $rt_{AVG} = (0 + 2 + 9 + 7) / 4 = 4.5$

A. Elkady

First-Come-First-Served (FCFS)

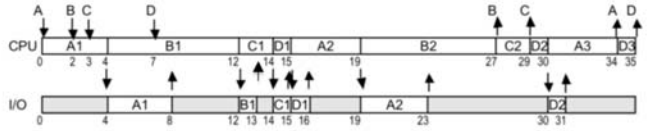


- Response time:
  - $rt_A = 0 - 0 = 0$
  - $rt_B = 4 - 2 = 2$
  - $rt_C = 12 - 3 = 9$
  - $rt_D = 14 - 7 = 7$

  $rt_{AVG} = (0 + 2 + 9 + 7) / 4 = 4.5$

A. Elkady

First-Come-First-Served (FCFS)



- Response time:
  - $rt_A = 0 - 0 = 0$
  - $rt_B = 4 - 2 = 2$
  - $rt_C = 12 - 3 = 9$
  - $rt_D = 14 - 7 = 7$
  - $rt_{AVG} = (0 + 2 + 9 + 7) / 4 = 4.5$

A. Elkady

What if we Knew the Future?

- Shortest Job First (SJF):
  - Run whatever job has the least amount of computation to do
- Shortest Remaining Time First (SRTF):
  - Preemptive version of SJF: if job arrives and has a shorter time to completion than the remaining time on the current job, immediately preempt CPU
    - but how do you now???
- Idea is to get short jobs out of the system
  - Big effect on short jobs, only small effect on long ones
  - Result is better average response time

A. Elkady

Interactive vs. batch scheduling

Batch

- First-Come-First-Served (FCFS)
- Shortest Job/Process First (SJF/SPF)
- Shortest Remaining Time First (SRTF)
- Priority (non-preemptive)

Interactive

- Round-Robin (RR) Priority (preemptive)
- Lottery

A. Elkady

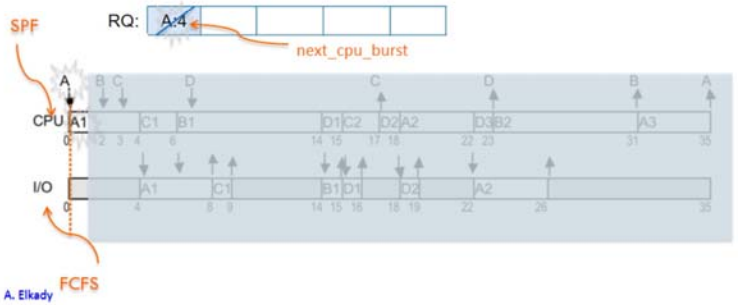
Shortest-Process-First (SPF)

- In this method, the processor is assigned to *the process with the smallest execution (processor burst) time.*
- This requires future knowledge of the execution time.
- In our examples, it is given as a table but actually these burst times are not known by the OS. So it makes prediction.
- One approach for this prediction is using the previous processor burst times for the processes in the ready queue and then the algorithm selects the shortest predicted next processor burst time.

A. Elkady

Shortest-Process-First (SPF)

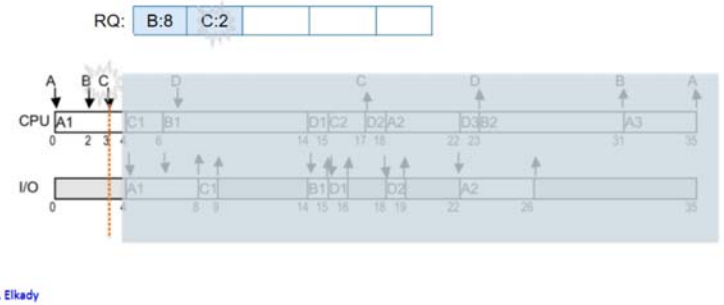
Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4	4	4	4	4
B	2	8	1	8	-	-
C	3	2	1	2	-	-
D	7	1	1	1	1	1



A. Elkady

Shortest-Process-First (SPF)

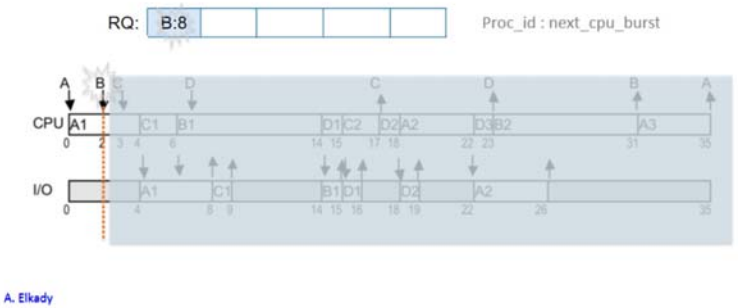
Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4	4	4	4	4
B	2	8	1	8	-	-
C	3	2	1	2	-	-
D	7	1	1	1	1	1



A. Elkady

Shortest-Process-First (SPF)

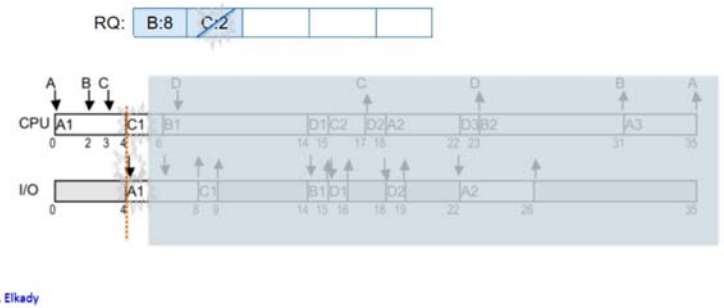
Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4	4	4	4	4
B	2	8	1	8	-	-
C	3	2	1	2	-	-
D	7	1	1	1	1	1



A. Elkady

Shortest-Process-First (SPF)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4	4	4	4	4
B	2	8	1	8	-	-
C	3	2	1	2	-	-
D	7	1	1	1	1	1

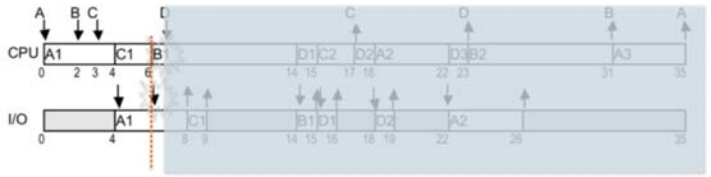


A. Elkady

Shortest-Process-First (SPF)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4	4	4	4	4
B	2	8	1	8	-	-
C	3	2	1	2	-	-
D	7	1	1	1	1	1

RQ: B:8

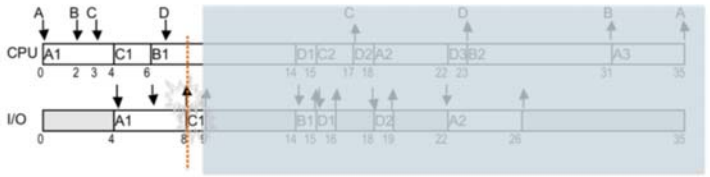


A. Elkady

Shortest-Process-First (SPF)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4	4	4	4	4
B	2	8	1	8	-	-
C	3	2	1	2	-	-
D	7	1	1	1	1	1

RQ: D:1 A:4

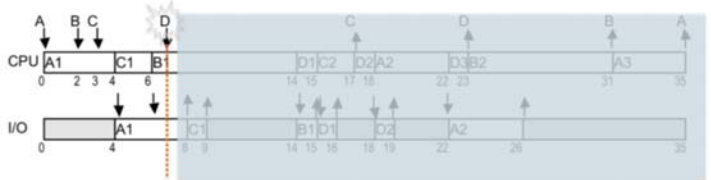


A. Elkady

Shortest-Process-First (SPF)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4	4	4	4	4
B	2	8	1	8	-	-
C	3	2	1	2	-	-
D	7	1	1	1	1	1

RQ: D:1

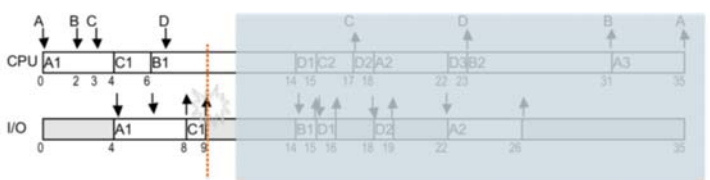


A. Elkady

Shortest-Process-First (SPF)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4	4	4	4	4
B	2	8	1	8	-	-
C	3	2	1	2	-	-
D	7	1	1	1	1	1

RQ: D:1 A:4 C:2

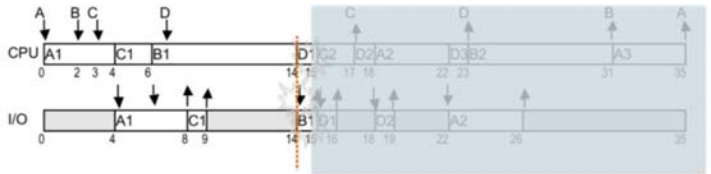


A. Elkady

Shortest-Process-First (SPF)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4	4	4	4	4
B	2	8	1	8	-	-
C	3	2	1	2	-	-
D	7	1	1	1	1	1

RQ: D:1 A:4 C:2

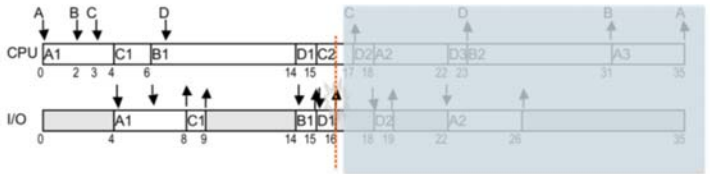


A. Elkady

Shortest-Process-First (SPF)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4	4	4	4	4
B	2	8	1	8	-	-
C	3	2	1	2	-	-
D	7	1	1	1	1	1

RQ: A:4 B:8 D:1

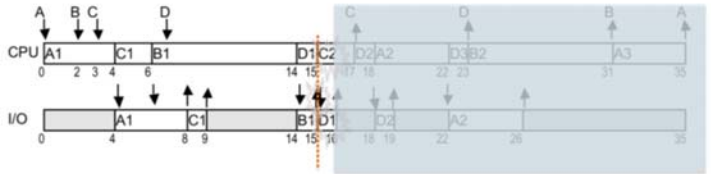


A. Elkady

Shortest-Process-First (SPF)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4	4	4	4	4
B	2	8	1	8	-	-
C	3	2	1	2	-	-
D	7	1	1	1	1	1

RQ: A:4 C:2 B:8

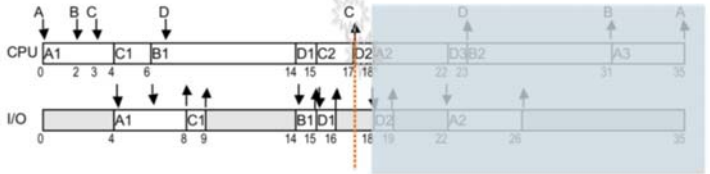


A. Elkady

Shortest-Process-First (SPF)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4	4	4	4	4
B	2	8	1	8	-	-
C	3	2	1	2	-	-
D	7	1	1	1	1	1

RQ: A:4 B:8 D:1

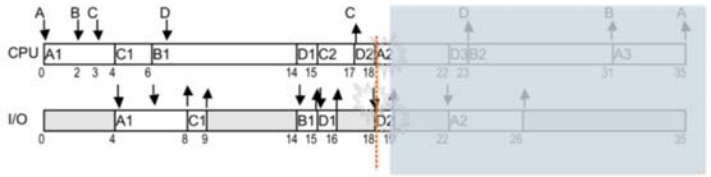


A. Elkady

Shortest-Process-First (SPF)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4	4	4	4	4
B	2	8	1	8	-	-
C	3	2	1	2	-	-
D	7	1	1	1	1	1

RQ: A:4 B:8

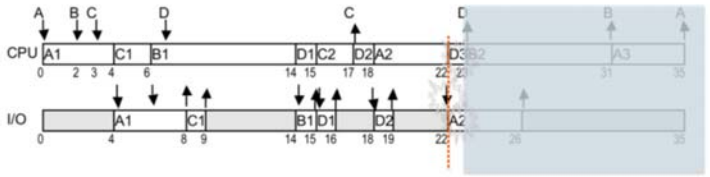


A. Elkady

Shortest-Process-First (SPF)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4	4	4	4	4
B	2	8	1	8	-	-
C	3	2	1	2	-	-
D	7	1	1	1	1	1

RQ: B:8 D:1

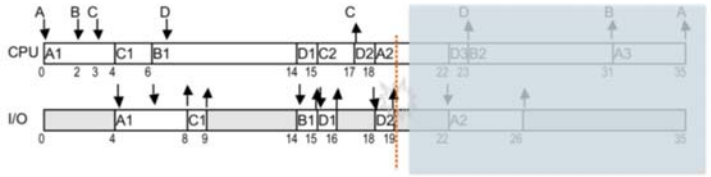


A. Elkady

Shortest-Process-First (SPF)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4	4	4	4	4
B	2	8	1	8	-	-
C	3	2	1	2	-	-
D	7	1	1	1	1	1

RQ: B:8 D:1

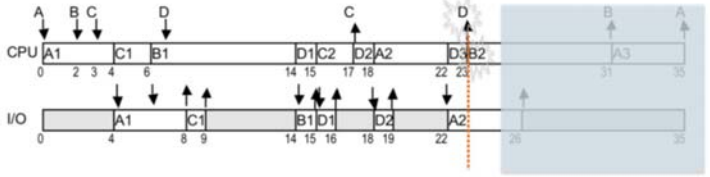


A. Elkady

Shortest-Process-First (SPF)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4	4	4	4	4
B	2	8	1	8	-	-
C	3	2	1	2	-	-
D	7	1	1	1	1	1

RQ: B:8

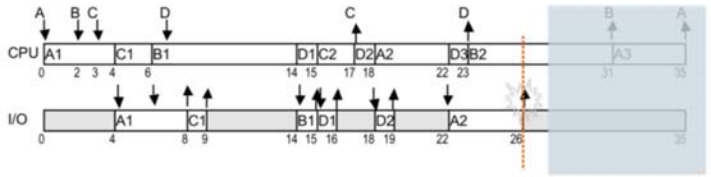


A. Elkady

Shortest-Process-First (SPF)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4	4	4	4	4
B	2	8	1	8	-	-
C	3	2	1	2	-	-
D	7	1	1	1	1	1

RQ: A:4

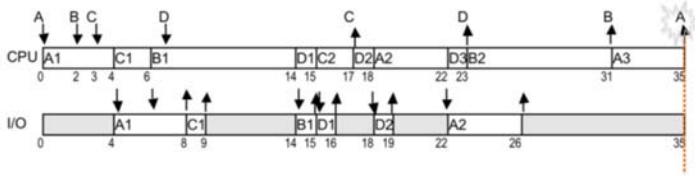


A. Elkady

Shortest-Process-First (SPF)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4	4	4	4	4
B	2	8	1	8	-	-
C	3	2	1	2	-	-
D	7	1	1	1	1	1

RQ:

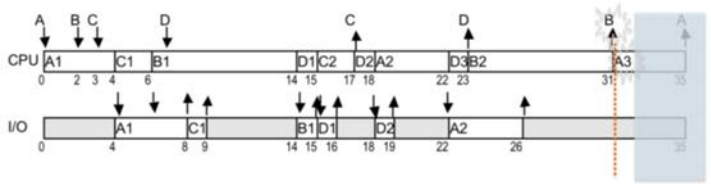


A. Elkady

Shortest-Process-First (SPF)

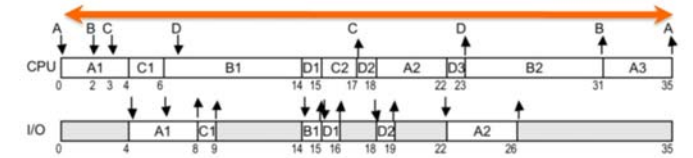
Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4	4	4	4	4
B	2	8	1	8	-	-
C	3	2	1	2	-	-
D	7	1	1	1	1	1

RQ: ~~A:4~~



A. Elkady

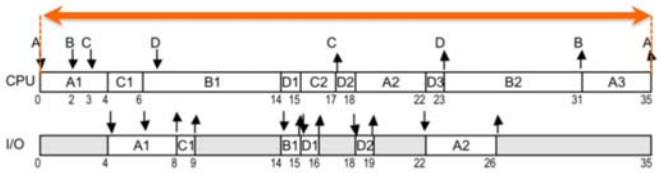
Shortest-Process-First (SPF)



- Processor utilization = (35 / 35) * 100 = 100 %
- Throughput = 4 / 35 = 0.11

A. Elkady

Shortest-Process-First (SPF)

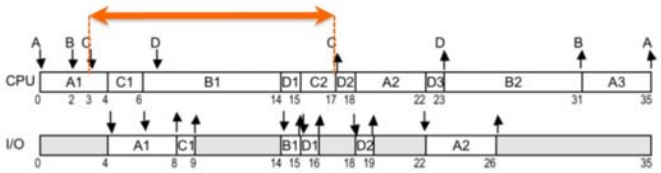


- Turn around time:
  - $tat_A = 35 - 0 = 35$
  - $tat_B = 31 - 2 = 29$
  - $tat_C = 17 - 3 = 14$
  - $tat_D = 23 - 7 = 16$

  $tat_{AVG} = (35 + 29 + 15 + 16) / 4 = 23.5$

A. Elkady

Shortest-Process-First (SPF)

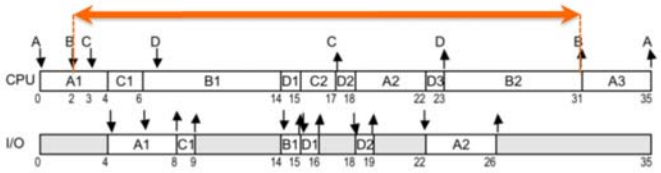


- Turn around time:
  - $tat_A = 35 - 0 = 35$
  - $tat_B = 31 - 2 = 29$
  - $tat_C = 17 - 3 = 14$
  - $tat_D = 23 - 7 = 16$

  $tat_{AVG} = (35 + 29 + 15 + 16) / 4 = 23.5$

A. Elkady

Shortest-Process-First (SPF)

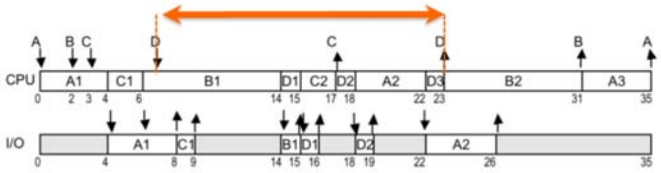


- Turn around time:
  - $tat_A = 35 - 0 = 35$
  - $tat_B = 31 - 2 = 29$
  - $tat_C = 17 - 3 = 14$
  - $tat_D = 23 - 7 = 16$

  $tat_{AVG} = (35 + 29 + 15 + 16) / 4 = 23.5$

A. Elkady

Shortest-Process-First (SPF)

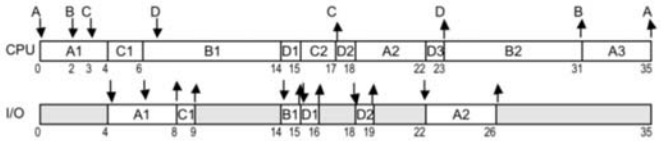


- Turn around time:
  - $tat_A = 35 - 0 = 35$
  - $tat_B = 31 - 2 = 29$
  - $tat_C = 17 - 3 = 14$
  - $tat_D = 23 - 7 = 16$

  $tat_{AVG} = (35 + 29 + 15 + 16) / 4 = 23.5$

A. Elkady

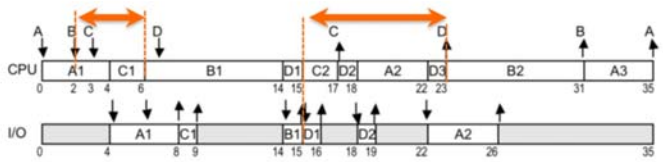
Shortest-Process-First (SPF)



- Turn around time:
  - $tat_A = 35 - 0 = 35$
  - $tat_B = 31 - 2 = 29$
  - $tat_C = 17 - 3 = 14$
  - $tat_D = 23 - 7 = 16$
  - $tat_{AVG} = (35 + 29 + 15 + 16) / 4 = 23.5$

A. Elkady

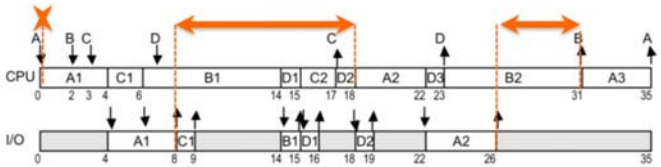
Shortest-Process-First (SPF)



- Waiting time:
  - $wt_A = (0 - 0) + (18 - 8) + (31 - 26) = 15$
  - $wt_B = (6 - 2) + (23 - 15) = 12$
  - $wt_C = (4 - 3) + (15 - 9) = 7$
  - $wt_D = (14 - 7) + (17 - 16) + (22 - 19) = 11$
  - $wt_{AVG} = (15 + 12 + 7 + 11) / 4 = 11.25$

A. Elkady

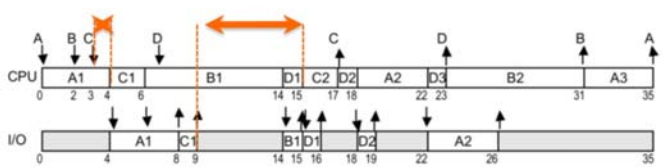
Shortest-Process-First (SPF)



- Waiting time:
  - $wt_A = (0 - 0) + (18 - 8) + (31 - 26) = 15$
  - $wt_B = (6 - 2) + (23 - 15) = 12$
  - $wt_C = (4 - 3) + (15 - 9) = 7$
  - $wt_D = (14 - 7) + (17 - 16) + (22 - 19) = 11$
  - $wt_{AVG} = (15 + 12 + 7 + 11) / 4 = 11.25$

A. Elkady

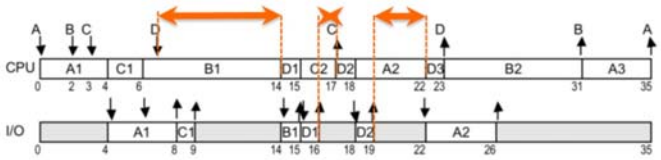
Shortest-Process-First (SPF)



- Waiting time:
  - $wt_A = (0 - 0) + (18 - 8) + (31 - 26) = 15$
  - $wt_B = (6 - 2) + (23 - 15) = 12$
  - $wt_C = (4 - 3) + (15 - 9) = 7$
  - $wt_D = (14 - 7) + (17 - 16) + (22 - 19) = 11$
  - $wt_{AVG} = (15 + 12 + 7 + 11) / 4 = 11.25$

A. Elkady

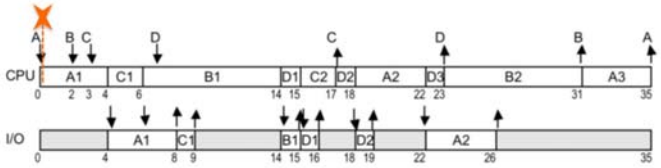
Shortest-Process-First (SPF)



- Waiting time:
  - $wt_A = (0 - 0) + (18 - 8) + (31 - 26) = 15$
  - $wt_B = (6 - 2) + (23 - 15) = 12$
  - $wt_C = (4 - 3) + (15 - 9) = 7$
  - $wt_D = (14 - 7) + (17 - 16) + (22 - 19) = 11$
  - $wt_{AVG} = (15 + 12 + 7 + 11) / 4 = 11.25$

A. Elkady

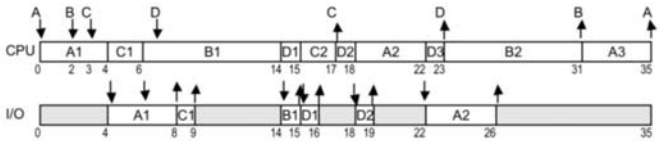
Shortest-Process-First (SPF)



- Response time:
  - $rt_A = 0 - 0 = 0$
  - $rt_B = 6 - 2 = 4$
  - $rt_C = 4 - 3 = 1$
  - $rt_D = 14 - 7 = 7$
  - $rt_{AVG} = (0 + 4 + 1 + 7) / 4 = 3$

A. Elkady

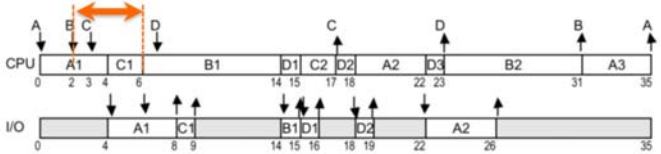
Shortest-Process-First (SPF)



- Waiting time:
  - $wt_A = (0 - 0) + (18 - 8) + (31 - 26) = 15$
  - $wt_B = (6 - 2) + (23 - 15) = 12$
  - $wt_C = (4 - 3) + (15 - 9) = 7$
  - $wt_D = (14 - 7) + (17 - 16) + (22 - 19) = 11$
  - $wt_{AVG} = (15 + 12 + 7 + 11) / 4 = 11.25$

A. Elkady

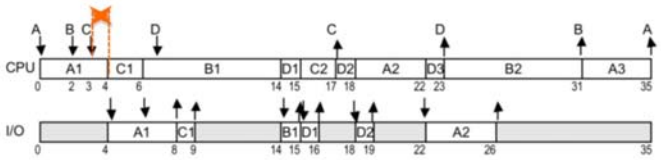
Shortest-Process-First (SPF)



- Response time:
  - $rt_A = 0 - 0 = 0$
  - $rt_B = 6 - 2 = 4$
  - $rt_C = 4 - 3 = 1$
  - $rt_D = 14 - 7 = 7$
  - $rt_{AVG} = (0 + 4 + 1 + 7) / 4 = 3$

A. Elkady

Shortest-Process-First (SPF)

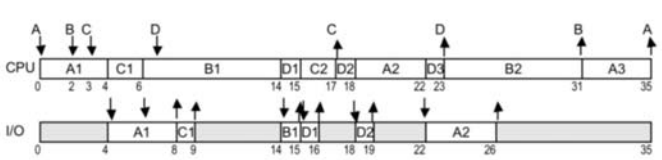


- Response time:
  - $rt_A = 0 - 0 = 0$
  - $rt_B = 6 - 2 = 4$
  - $rt_C = 4 - 3 = 1$
  - $rt_D = 14 - 7 = 7$

  $rt_{AVG} = (0 + 4 + 1 + 7) / 4 = 3$

A. Elkady

Shortest-Process-First (SPF)

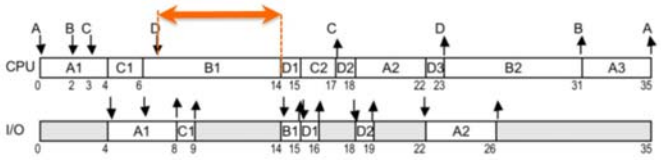


- Response time:
  - $rt_A = 0 - 0 = 0$
  - $rt_B = 6 - 2 = 4$
  - $rt_C = 4 - 3 = 1$
  - $rt_D = 14 - 7 = 7$

  $rt_{AVG} = (0 + 4 + 1 + 7) / 4 = 3$

A. Elkady

Shortest-Process-First (SPF)

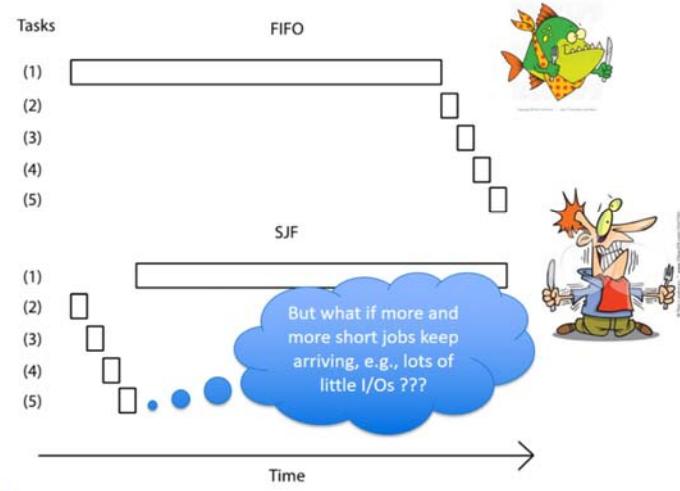


- Response time:
  - $rt_A = 0 - 0 = 0$
  - $rt_B = 6 - 2 = 4$
  - $rt_C = 4 - 3 = 1$
  - $rt_D = 14 - 7 = 7$

  $rt_{AVG} = (0 + 4 + 1 + 7) / 4 = 3$

A. Elkady

FIFO vs. SJF



A. Elkady

Interactive vs. batch scheduling

Batch

- First-Come-First-Served (FCFS)
- Shortest Job/Process First (SJF/SPF)
- Shortest Remaining Time First (SRTF)
- Priority (non-preemptive)

Interactive

- Round-Robin (RR) Priority (preemptive)
- Lottery

A. Elkady

Shortest-Remaining-Time-First (SRTF)

- SPF algorithm can be modified to be preemptive.
- Assume while one process is executing on the processor, another process arrives.
- The new process may have a predicted next processor burst time shorter than what is left of the currently executing process.
- If the SPF algorithm is preemptive, the currently executing process will preempt the processor and the new process will start executing.
- The modified SPF algorithm is named as Preemptive SPF or Shortest-Remaining-Time-First (SRTF) algorithm.

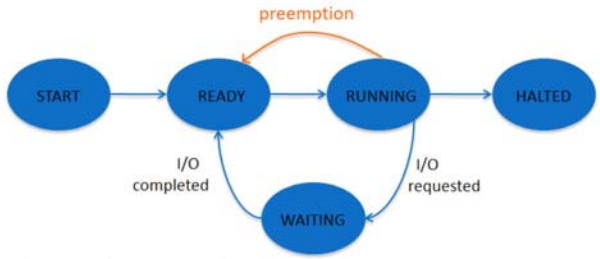
A. Elkady

Shortest-Remaining-Time-First (SRTF)

- The scheduling algorithms we discussed so far are all **non-preemptive** algorithms.
- That is, once a process grabs the processor, it keeps the processor until it terminates or it requests I/O.
- To deal with this problem (if so), **preemptive algorithms** are developed.
- In this type of algorithms, at some time instant, the process being executed is forced to preempt CPU in order to execute a new selected process.
- The preemption conditions are up to the algorithm design.

A. Elkady

Shortest-Remaining-Time-First (SRTF)

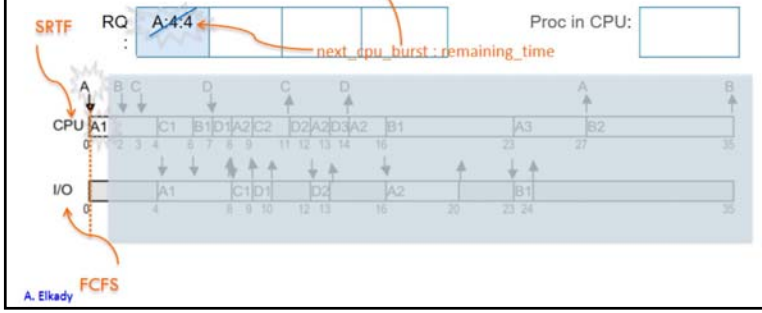


Start : The process has just arrived.
Ready : The process is waiting to grab the processor.
Running : The process has been allocated by the processor.
Waiting : The process is doing I/O work or blocked.
Halted : The process has finished and is about to leave the system

A. Elkady

Shortest-Remaining-Time-First (SRTF)

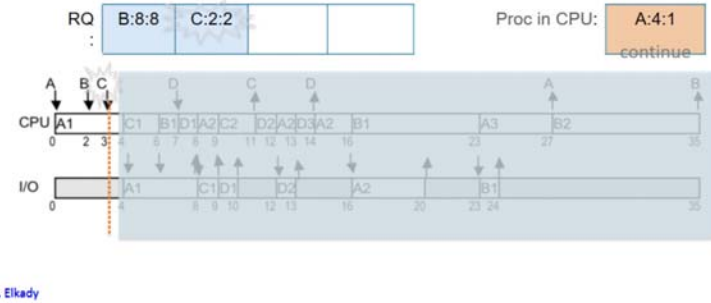
Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4 : 4	4	4 : 4	4	4 : 4
B	2	8 : 8	1	8 : 8	-	-
C	3	2 : 2	1	2 : 2	-	-
D	7	1 : 1	1	1 : 1	1	1 : 1



A. Elkady

Shortest-Remaining-Time-First (SRTF)

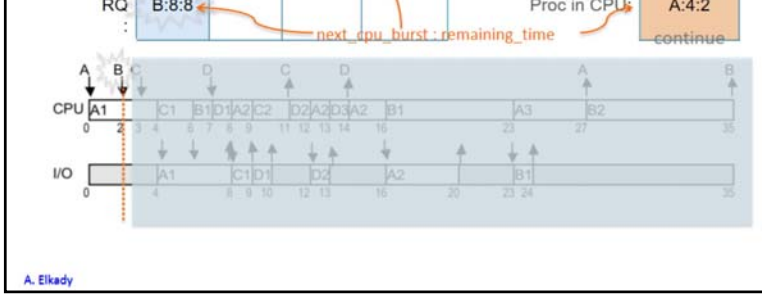
Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4 : 1	4	4 : 4	4	4 : 4
B	2	8 : 8	1	8 : 8	-	-
C	3	2 : 2	1	2 : 2	-	-
D	7	1 : 1	1	1 : 1	1	1 : 1



A. Elkady

Shortest-Remaining-Time-First (SRTF)

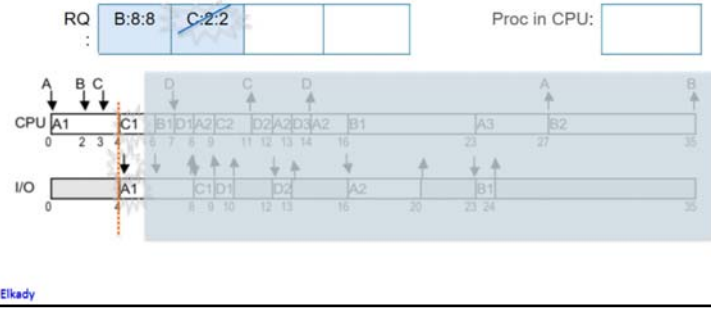
Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4 : 2	4	4 : 4	4	4 : 4
B	2	8 : 8	1	8 : 8	-	-
C	3	2 : 2	1	2 : 2	-	-
D	7	1 : 1	1	1 : 1	1	1 : 1



A. Elkady

Shortest-Remaining-Time-First (SRTF)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4 : 0	4	4 : 4	4	4 : 4
B	2	8 : 8	1	8 : 8	-	-
C	3	2 : 2	1	2 : 2	-	-
D	7	1 : 1	1	1 : 1	1	1 : 1

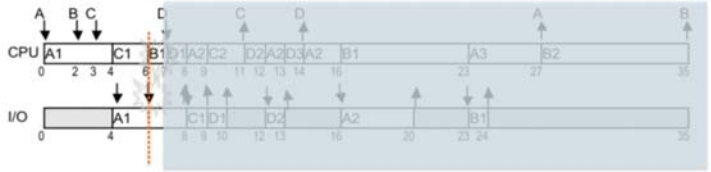


A. Elkady

Shortest-Remaining-Time-First (SRTF)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4 : 0	4	4 : 4	4	4 : 4
B	2	8 : 8	1	8 : 8	-	-
C	3	2 : 0	1	2 : 2	-	-
D	7	1 : 1	1	1 : 1	1	1 : 1

RQ : B:8:8 Proc in CPU:

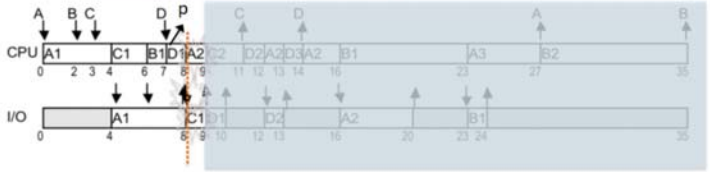


A. Elkady

Shortest-Remaining-Time-First (SRTF)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4 : 0	4	4 : 4	4	4 : 4
B	2	8 : 7	1	8 : 8	-	-
C	3	2 : 0	1	2 : 2	-	-
D	7	1 : 0	1	1 : 1	1	1 : 1

RQ : B:8:7 A:4:4 Proc in CPU:



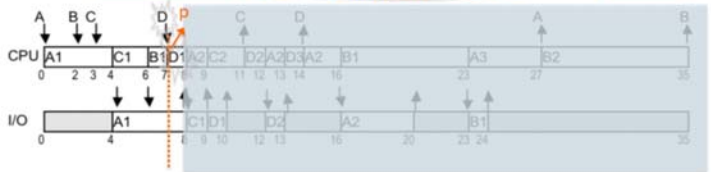
A. Elkady

Shortest-Remaining-Time-First (SRTF)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4 : 0	4	4 : 4	4	4 : 4
B	2	8 : 7	1	8 : 8	-	-
C	3	2 : 0	1	2 : 2	-	-
D	7	1 : 1	1	1 : 1	1	1 : 1

RQ : D:1:1 B:8:7 Proc in CPU: B:8:7

preemption



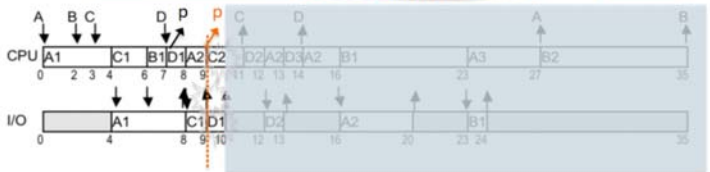
A. Elkady

Shortest-Remaining-Time-First (SRTF)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4 : 0	4	4 : 3	4	4 : 4
B	2	8 : 7	1	8 : 8	-	-
C	3	2 : 0	1	2 : 2	-	-
D	7	1 : 0	1	1 : 1	1	1 : 1

RQ : B:8:7 C:2:2 A:4:3 Proc in CPU: A:4:3

preemption



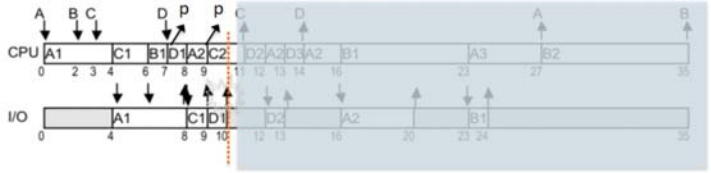
A. Elkady

Shortest-Remaining-Time-First (SRTF)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4 : 0	4	4 : 3	4	4 : 4
B	2	8 : 7	1	8 : 8	-	-
C	3	2 : 0	1	2 : 1	-	-
D	7	1 : 0	1	1 : 1	1	1 : 1

RQ : B:8:7 | A:4:3 | D:1:1

Proc in CPU: C:2:1 continues



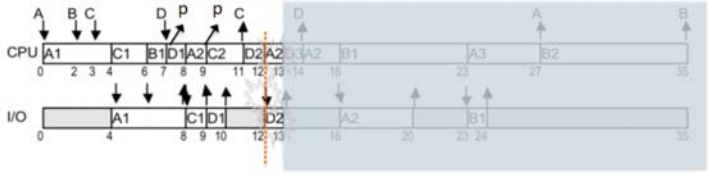
A. Elkady

Shortest-Remaining-Time-First (SRTF)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4 : 0	4	4 : 3	4	4 : 4
B	2	8 : 7	1	8 : 8	-	-
C	3	2 : 0	1	2 : 0	-	-
D	7	1 : 0	1	1 : 0	1	1 : 1

RQ : B:8:7 | A:4:3

Proc in CPU:



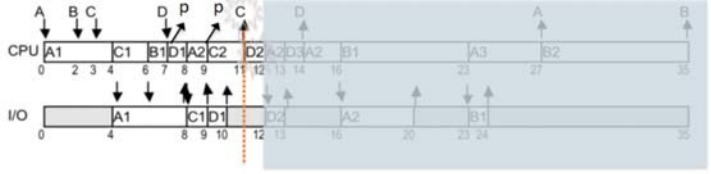
A. Elkady

Shortest-Remaining-Time-First (SRTF)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4 : 0	4	4 : 3	4	4 : 4
B	2	8 : 7	1	8 : 8	-	-
C	3	2 : 0	1	2 : 0	-	-
D	7	1 : 0	1	1 : 1	1	1 : 1

RQ : B:8:7 | A:4:3 | D:1:1

Proc in CPU:



A. Elkady

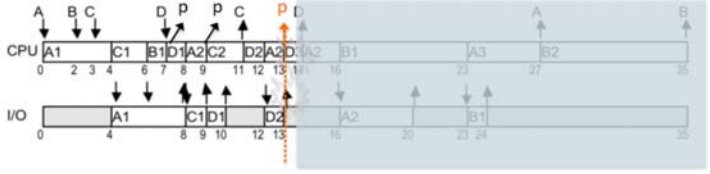
Shortest-Remaining-Time-First (SRTF)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4 : 0	4	4 : 2	4	4 : 4
B	2	8 : 7	1	8 : 8	-	-
C	3	2 : 0	1	2 : 0	-	-
D	7	1 : 0	1	1 : 0	1	1 : 1

RQ : B:8:7 | D:4:1 | A:4:2

Proc in CPU: A:4:2

preemption



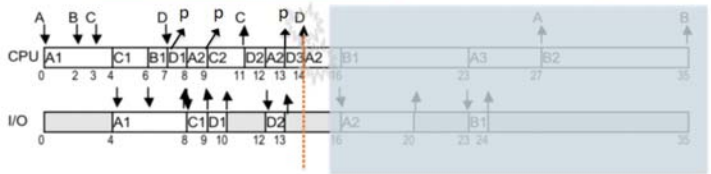
A. Elkady

Shortest-Remaining-Time-First (SRTF)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4 : 0	4	4 : 2	4	4 : 4
B	2	8 : 7	1	8 : 8	-	-
C	3	2 : 0	1	2 : 0	-	-
D	7	1 : 0	1	1 : 0	1	1 : 0

RQ : B:8:7 A:4:2

Proc in CPU:



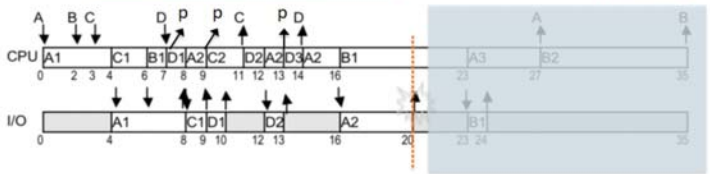
A. Elkady

Shortest-Remaining-Time-First (SRTF)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4 : 0	4	4 : 0	4	4 : 4
B	2	8 : 3	1	8 : 8	-	-
C	3	2 : 0	1	2 : 0	-	-
D	7	1 : 0	1	1 : 0	1	1 : 0

RQ : A:4:4

Proc in CPU: B:8:3 continues



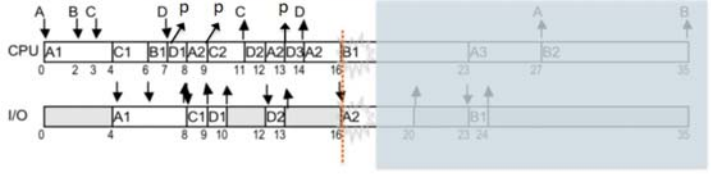
A. Elkady

Shortest-Remaining-Time-First (SRTF)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4 : 0	4	4 : 0	4	4 : 4
B	2	7 : 7	1	8 : 8	-	-
C	3	2 : 0	1	2 : 0	-	-
D	7	1 : 0	1	1 : 0	1	1 : 0

RQ : B:8:7

Proc in CPU:



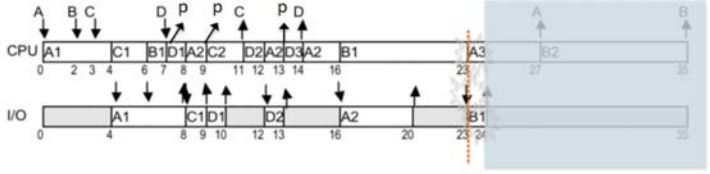
A. Elkady

Shortest-Remaining-Time-First (SRTF)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4 : 0	4	4 : 0	4	4 : 4
B	2	8 : 0	1	8 : 8	-	-
C	3	2 : 0	1	2 : 0	-	-
D	7	1 : 0	1	1 : 0	1	1 : 0

RQ : A:4:4

Proc in CPU:



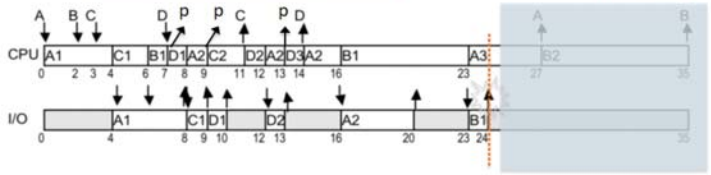
A. Elkady

Shortest-Remaining-Time-First (SRTF)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4 : 0	4	4 : 0	4	4 : 3
B	2	8 : 0	1	8 : 8	-	-
C	3	2 : 0	1	2 : 0	-	-
D	7	1 : 0	1	1 : 0	1	1 : 0

RQ : B:8:8

Proc in CPU: A:4:3 continues



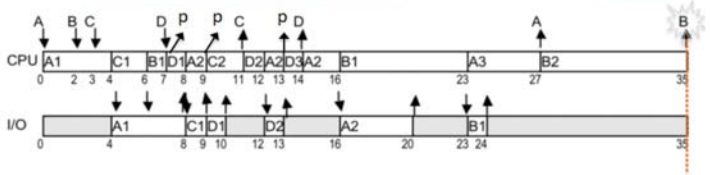
A. Elkady

Shortest-Remaining-Time-First (SRTF)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4 : 0	4	4 : 0	4	4 : 0
B	2	8 : 0	1	8 : 0	-	-
C	3	2 : 0	1	2 : 0	-	-
D	7	1 : 0	1	1 : 0	1	1 : 0

RQ :

Proc in CPU:



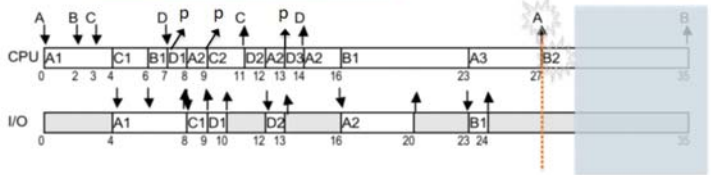
A. Elkady

Shortest-Remaining-Time-First (SRTF)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4 : 0	4	4 : 0	4	4 : 0
B	2	8 : 0	1	8 : 1	-	-
C	3	2 : 0	1	2 : 0	-	-
D	7	1 : 0	1	1 : 0	1	1 : 0

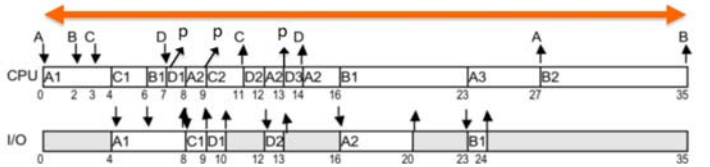
RQ : B:8:8

Proc in CPU:



A. Elkady

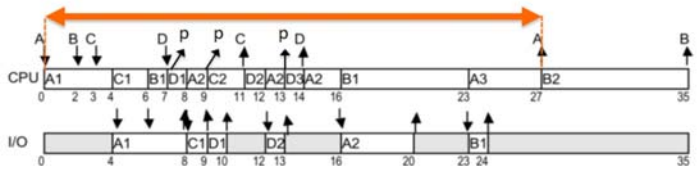
Shortest-Remaining-Time-First (SRTF)



- Processor utilization = (35 / 35) * 100 = 100 %
- Throughput = 4 / 35 = 0.11

A. Elkady

Shortest-Remaining-Time-First (SRTF)

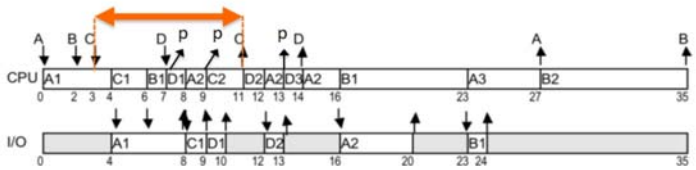


- Turn around time:
  - $tat_A = 27 - 0 = 27$
  - $tat_B = 35 - 2 = 33$
  - $tat_C = 11 - 3 = 8$
  - $tat_D = 14 - 7 = 7$

  $tat_{AVG} = (27 + 33 + 8 + 7) / 4 = 18.75$

A. Elkady

Shortest-Remaining-Time-First (SRTF)

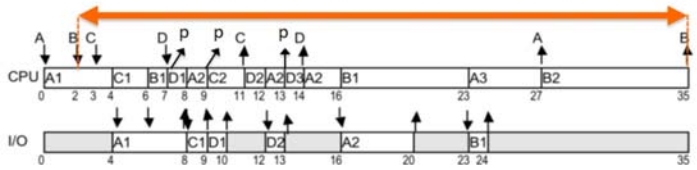


- Turn around time:
  - $tat_A = 27 - 0 = 27$
  - $tat_B = 35 - 2 = 33$
  - $tat_C = 11 - 3 = 8$
  - $tat_D = 14 - 7 = 7$

  $tat_{AVG} = (27 + 33 + 8 + 7) / 4 = 18.75$

A. Elkady

Shortest-Remaining-Time-First (SRTF)

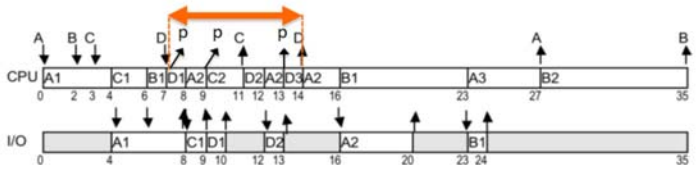


- Turn around time:
  - $tat_A = 27 - 0 = 27$
  - $tat_B = 35 - 2 = 33$
  - $tat_C = 11 - 3 = 8$
  - $tat_D = 14 - 7 = 7$

  $tat_{AVG} = (27 + 33 + 8 + 7) / 4 = 18.75$

A. Elkady

Shortest-Remaining-Time-First (SRTF)

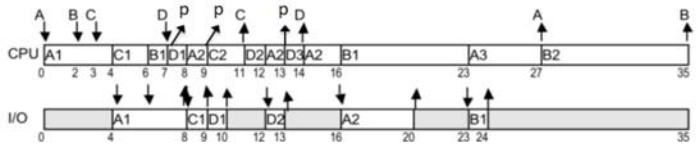


- Turn around time:
  - $tat_A = 27 - 0 = 27$
  - $tat_B = 35 - 2 = 33$
  - $tat_C = 11 - 3 = 8$
  - $tat_D = 14 - 7 = 7$

  $tat_{AVG} = (27 + 33 + 8 + 7) / 4 = 18.75$

A. Elkady

Shortest-Remaining-Time-First (SRTF)

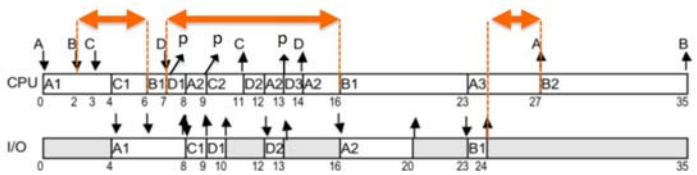


- Turn around time:
  - $tat_A = 27 - 0 = 27$
  - $tat_B = 35 - 2 = 33$
  - $tat_C = 11 - 3 = 8$
  - $tat_D = 14 - 7 = 7$

  $tat_{AVG} = (27 + 33 + 8 + 7) / 4 = 18.75$

A. Elkady

Shortest-Remaining-Time-First (SRTF)

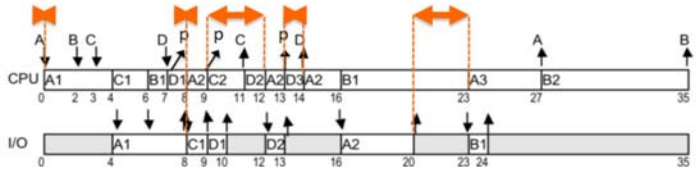


- Waiting time:
  - $wt_A = (0 - 0) + (8 - 8) + (12 - 9) + (14 - 13) + (23 - 20) = 7$
  - $wt_B = (6 - 2) + (16 - 7) + (27-24) = 16$
  - $wt_C = (4 - 3) + (9 - 9) = 1$
  - $wt_D = (7 - 7) + (11 - 10) + (13 - 13) = 1$

  $wt_{AVG} = (7 + 16 + 1 + 1) / 4 = 6.25$

A. Elkady

Shortest-Remaining-Time-First (SRTF)

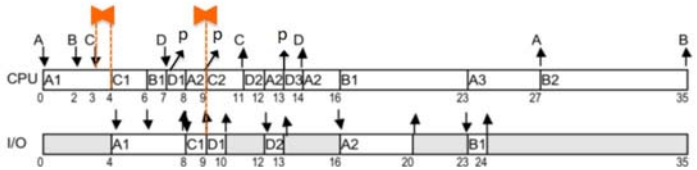


- Waiting time:
  - $wt_A = (0 - 0) + (8 - 8) + (12 - 9) + (14 - 13) + (23 - 20) = 7$
  - $wt_B = (6 - 2) + (16 - 7) + (27-24) = 16$
  - $wt_C = (4 - 3) + (9 - 9) = 1$
  - $wt_D = (7 - 7) + (11 - 10) + (13 - 13) = 1$

  $wt_{AVG} = (7 + 16 + 1 + 1) / 4 = 6.25$

A. Elkady

Shortest-Remaining-Time-First (SRTF)

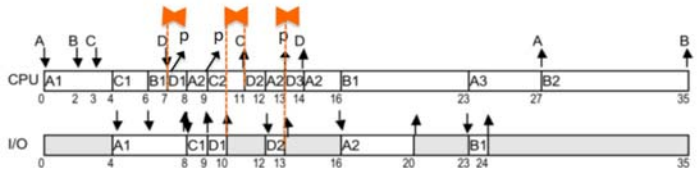


- Waiting time:
  - $wt_A = (0 - 0) + (8 - 8) + (12 - 9) + (14 - 13) + (23 - 20) = 7$
  - $wt_B = (6 - 2) + (16 - 7) + (27-24) = 16$
  - $wt_C = (4 - 3) + (9 - 9) = 1$
  - $wt_D = (7 - 7) + (11 - 10) + (13 - 13) = 1$

  $wt_{AVG} = (7 + 16 + 1 + 1) / 4 = 6.25$

A. Elkady

Shortest-Remaining-Time-First (SRTF)

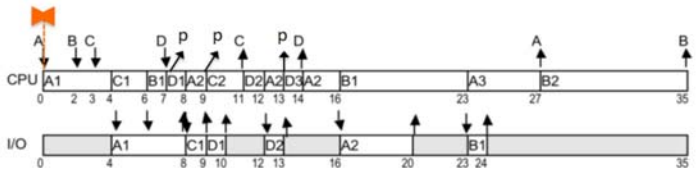


- Waiting time:
  - $wt_A = (0-0) + (8-8) + (12-9) + (14-13) + (23-20) = 7$
  - $wt_B = (6-2) + (16-7) + (27-24) = 16$
  - $wt_C = (4-3) + (9-9) = 1$
  - $wt_D = (7-7) + (11-10) + (13-13) = 1$

  $wt_{AVG} = (7 + 16 + 1 + 1) / 4 = 6.25$

A. Elkady

Shortest-Remaining-Time-First (SRTF)

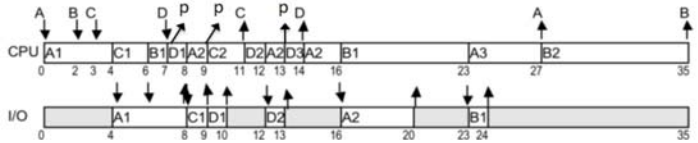


- Response time:
  - $rt_A = 0 - 0 = 0$
  - $rt_B = 6 - 2 = 4$
  - $rt_C = 4 - 3 = 1$
  - $rt_D = 7 - 7 = 0$

  $rt_{AVG} = (0 + 4 + 1 + 0) / 4 = 1.25$

A. Elkady

Shortest-Remaining-Time-First (SRTF)

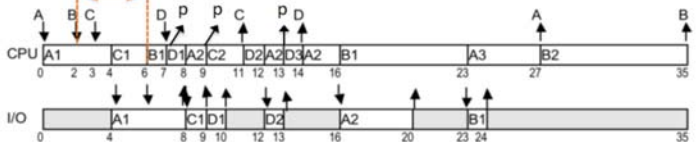


- Waiting time:
  - $wt_A = (0-0) + (8-8) + (12-9) + (14-13) + (23-20) = 7$
  - $wt_B = (6-2) + (16-7) + (27-24) = 16$
  - $wt_C = (4-3) + (9-9) = 1$
  - $wt_D = (7-7) + (11-10) + (13-13) = 1$

  $wt_{AVG} = (7 + 16 + 1 + 1) / 4 = 6.25$

A. Elkady

Shortest-Remaining-Time-First (SRTF)

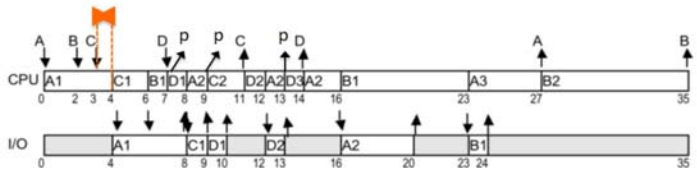


- Response time:
  - $rt_A = 0 - 0 = 0$
  - $rt_B = 6 - 2 = 4$
  - $rt_C = 4 - 3 = 1$
  - $rt_D = 7 - 7 = 0$

  $rt_{AVG} = (0 + 4 + 1 + 0) / 4 = 1.25$

A. Elkady

Shortest-Remaining-Time-First (SRTF)

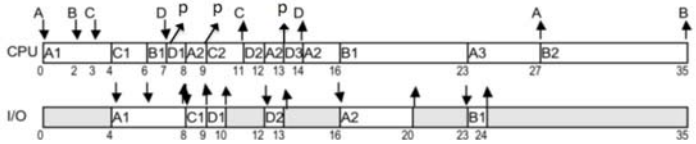


- Response time:
  - $rt_A = 0 - 0 = 0$
  - $rt_B = 6 - 2 = 4$
  - $rt_C = 4 - 3 = 1$
  - $rt_D = 7 - 7 = 0$

  $rt_{AVG} = (0 + 4 + 1 + 0) / 4 = 1.25$

A. Elkady

Shortest-Remaining-Time-First (SRTF)

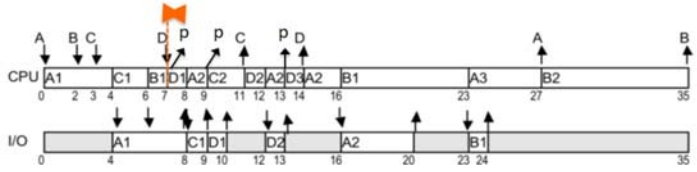


- Response time:
  - $rt_A = 0 - 0 = 0$
  - $rt_B = 6 - 2 = 4$
  - $rt_C = 4 - 3 = 1$
  - $rt_D = 7 - 7 = 0$

  $rt_{AVG} = (0 + 4 + 1 + 0) / 4 = 1.25$

A. Elkady

Shortest-Remaining-Time-First (SRTF)



- Response time:
  - $rt_A = 0 - 0 = 0$
  - $rt_B = 6 - 2 = 4$
  - $rt_C = 4 - 3 = 1$
  - $rt_D = 7 - 7 = 0$

  $rt_{AVG} = (0 + 4 + 1 + 0) / 4 = 1.25$

A. Elkady

Interactive vs. batch scheduling

Batch

- First-Come-First-Served (FCFS)
- Shortest Job/Process First (SJF/SPF)
- Shortest Remaining Time First (SRTF)
- Priority (non-preemptive)

Interactive

- Round-Robin (RR) Priority (preemptive)
- Lottery

A. Elkady

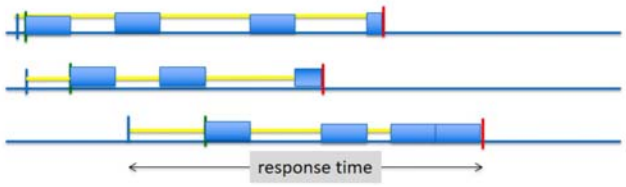
Round-Robin Scheduling (RRS)

- In RRS algorithm the ready queue is treated as a FIFO circular queue.
- The RRS traces the ready queue allocating the processor to each process for a time interval which is smaller than or equal to a predefined time called **time quantum (slice).**

A. Elkady

Round-Robin

- Each task gets a fixed amount of the resource (time quantum)
  - if does not complete, goes back into queue



- How large a time quantum?
  - Too short? Too long? Trade-offs?

A. Elkady

Round-Robin Scheduling (RRS)

- The OS using RRS, takes the first process from the ready queue, gives the processor to that process and sets a timer to interrupt when time quantum expires.
- If the process has a processor burst time smaller than the time quantum, then it releases the processor voluntarily, either by terminating or by issuing an I/O request.
- The OS then proceed with the next process in the ready queue.

A. Elkady

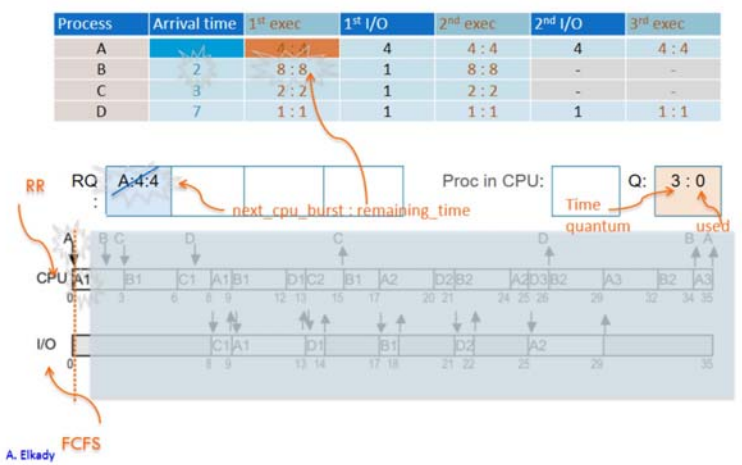
Round-Robin Scheduling (RRS)

- The performance of RRS depends heavily on the selected time quantum.
- Time quantum → ∞ ⇒ RRS becomes FCFS
- Time quantum → 0 ⇒ RRS becomes processor sharing (It acts as if each of the n processes has its own processor running at processor speed divided by n)
- However, if time quantum is too small then context switch time can not be ignored anymore.
- For an optimum time quantum, it can be selected to be greater than 80 % of processor bursts and to be greater than the context switching time.

A. Elkady

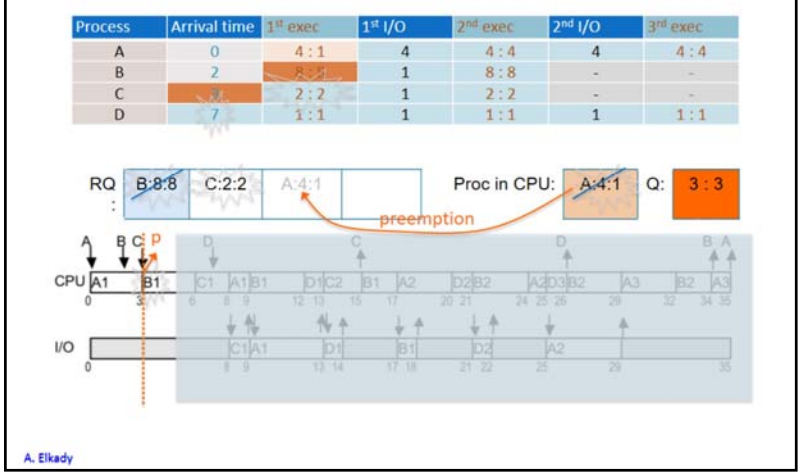
Round-Robin Scheduling (RRS)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4 : 4	4	4 : 4	4	4 : 4
B	2	8 : 8	1	8 : 8	-	-
C	3	2 : 2	1	2 : 2	-	-
D	7	1 : 1	1	1 : 1	1	1 : 1



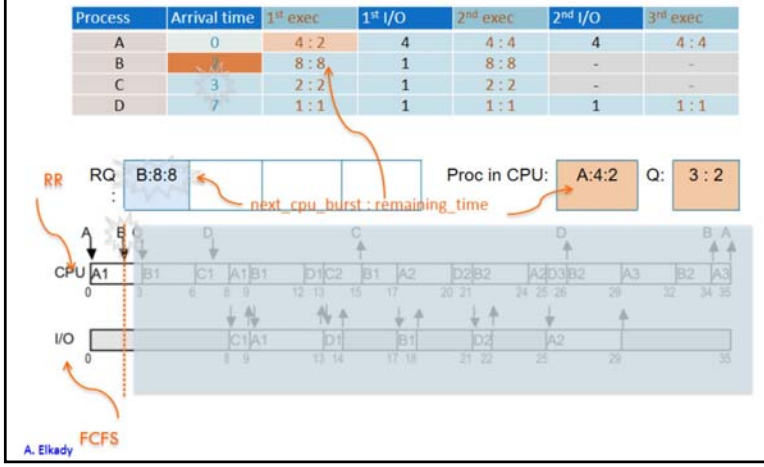
Round-Robin Scheduling (RRS)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4 : 1	4	4 : 4	4	4 : 4
B	2	8 : 8	1	8 : 8	-	-
C	3	2 : 2	1	2 : 2	-	-
D	7	1 : 1	1	1 : 1	1	1 : 1



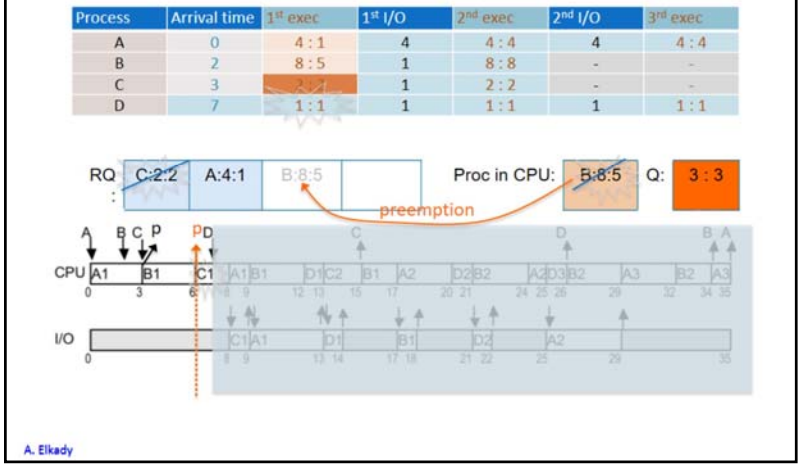
Round-Robin Scheduling (RRS)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4 : 2	4	4 : 4	4	4 : 4
B	2	8 : 8	1	8 : 8	-	-
C	3	2 : 2	1	2 : 2	-	-
D	7	1 : 1	1	1 : 1	1	1 : 1



Round-Robin Scheduling (RRS)

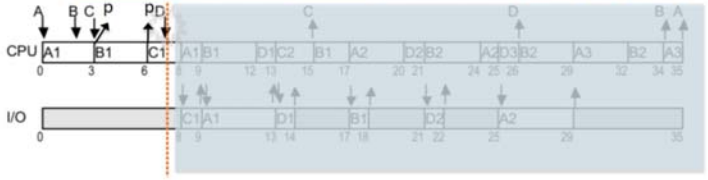
Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4 : 1	4	4 : 4	4	4 : 4
B	2	8 : 5	1	8 : 8	-	-
C	3	2 : 2	1	2 : 2	-	-
D	7	1 : 1	1	1 : 1	1	1 : 1



Round-Robin Scheduling (RRS)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4 : 1	4	4 : 4	4	4 : 4
B	2	8 : 5	1	8 : 8	-	-
C	3	2 : 1	1	2 : 2	-	-
D	7	1 : 1	1	1 : 1	1	1 : 1

RQ : A:4:1 | B:8:5 | D:1:1 | Proc in CPU: C:2:1 | Q: 3 : 1 expires

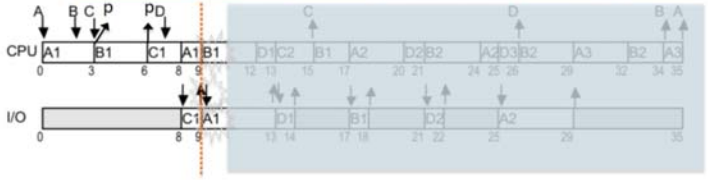


A. Elkady

Round-Robin Scheduling (RRS)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4 : 0	4	4 : 4	4	4 : 4
B	2	8 : 5	1	8 : 8	-	-
C	3	2 : 0	1	2 : 2	-	-
D	7	1 : 1	1	1 : 1	1	1 : 1

RQ : B:8:5 | D:1:1 | C:2:2 | Proc in CPU: | Q: 3 : 0

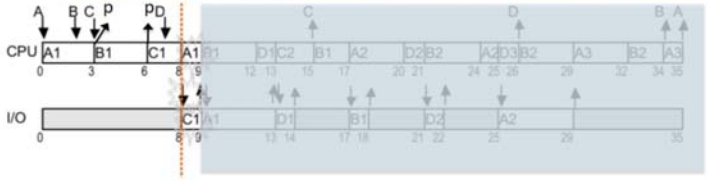


A. Elkady

Round-Robin Scheduling (RRS)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4 : 1	4	4 : 4	4	4 : 4
B	2	8 : 5	1	8 : 8	-	-
C	3	2 : 0	1	2 : 2	-	-
D	7	1 : 1	1	1 : 1	1	1 : 1

RQ : A:4:1 | B:8:5 | D:1:1 | Proc in CPU: | Q: 3 : 0



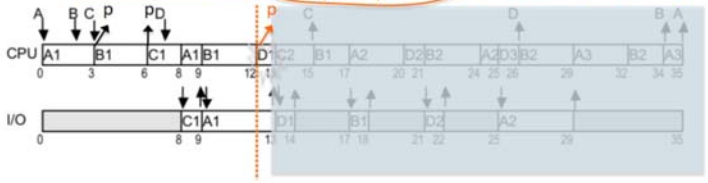
A. Elkady

Round-Robin Scheduling (RRS)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4 : 0	4	4 : 4	4	4 : 4
B	2	8 : 2	1	8 : 8	-	-
C	3	2 : 0	1	2 : 2	-	-
D	7	1 : 1	1	1 : 1	1	1 : 1

RQ : D:1:1 | C:2:2 | B:8:2 | Proc in CPU: B:8:2 | Q: 3 : 3

preemption

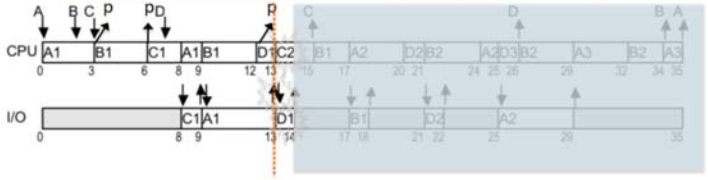


A. Elkady

Round-Robin Scheduling (RRS)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4 : 0	4	4 : 4	4	4 : 4
B	2	8 : 2	1	8 : 8	-	-
C	3	2 : 0	1	2 : 2	-	-
D	7	1 : 0	1	1 : 1	1	1 : 1

RQ : C:2:2 | B:8:2 | A:4:4 | Proc in CPU: | Q: 3 : 0

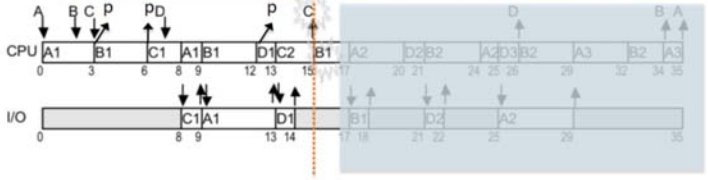


A. Elkady

Round-Robin Scheduling (RRS)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4 : 0	4	4 : 4	4	4 : 4
B	2	8 : 2	1	8 : 8	-	-
C	3	2 : 0	1	2 : 0	-	-
D	7	1 : 0	1	1 : 1	1	1 : 1

RQ : B:8:2 | A:4:4 | D:1:1 | Proc in CPU: | Q: 3 : 0

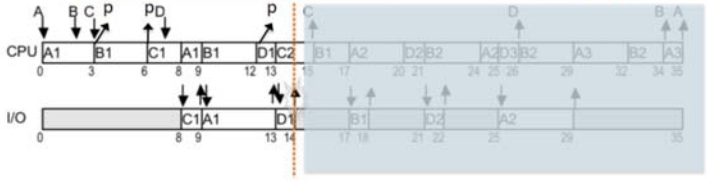


A. Elkady

Round-Robin Scheduling (RRS)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4 : 0	4	4 : 4	4	4 : 4
B	2	8 : 2	1	8 : 8	-	-
C	3	2 : 0	1	2 : 1	-	-
D	7	1 : 0	1	1 : 1	1	1 : 1

RQ : B:8:2 | A:4:4 | D:1:1 | Proc in CPU: C:2:1 | Q: 3 : 1 expires

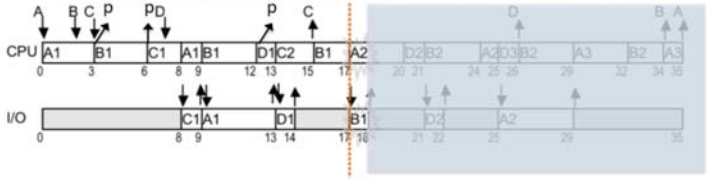


A. Elkady

Round-Robin Scheduling (RRS)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4 : 0	4	4 : 4	4	4 : 4
B	2	8 : 0	1	8 : 8	-	-
C	3	2 : 0	1	2 : 0	-	-
D	7	1 : 0	1	1 : 1	1	1 : 1

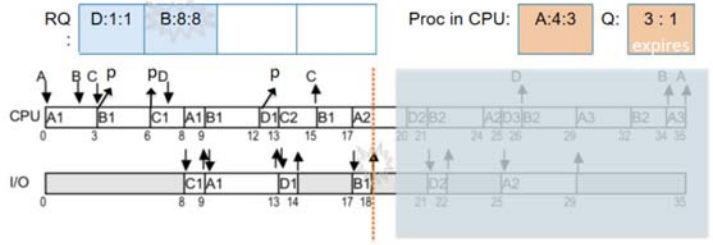
RQ : A:4:4 | D:1:1 | Proc in CPU: | Q: 3 : 0



A. Elkady

Round-Robin Scheduling (RRS)

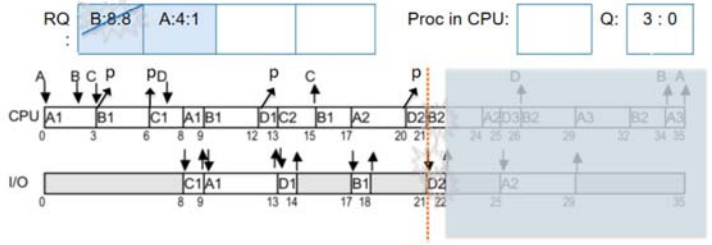
Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4 : 0	4	4 : 3	4	4 : 4
B	2	8 : 0	1	8 : 8	-	-
C	3	2 : 0	1	2 : 0	-	-
D	7	1 : 0	1	1 : 1	1	1 : 1



A. Elkady

Round-Robin Scheduling (RRS)

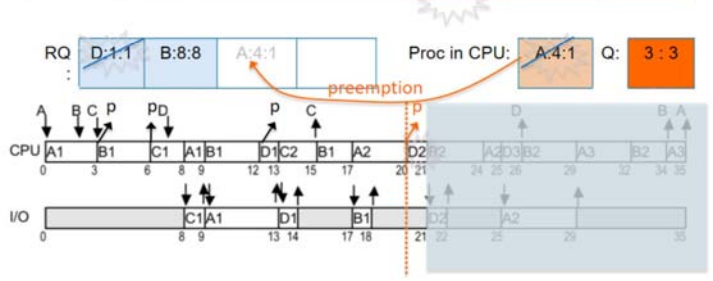
Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4 : 0	4	4 : 1	4	4 : 4
B	2	8 : 0	1	8 : 8	-	-
C	3	2 : 0	1	2 : 0	-	-
D	7	1 : 0	1	1 : 0	1	1 : 1



A. Elkady

Round-Robin Scheduling (RRS)

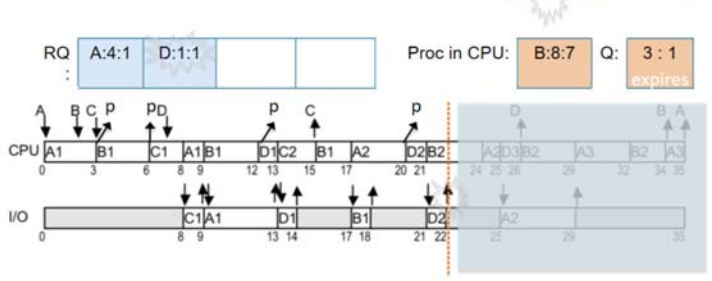
Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4 : 0	4	4 : 1	4	4 : 4
B	2	8 : 0	1	8 : 8	-	-
C	3	2 : 0	1	2 : 0	-	-
D	7	1 : 0	1	1 : 1	1	1 : 1



A. Elkady

Round-Robin Scheduling (RRS)

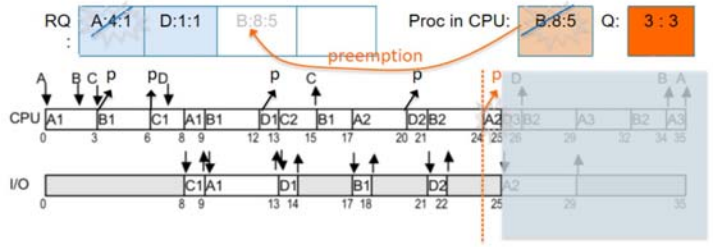
Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4 : 0	4	4 : 1	4	4 : 4
B	2	8 : 0	1	8 : 7	-	-
C	3	2 : 0	1	2 : 0	-	-
D	7	1 : 0	1	1 : 0	1	1 : 1



A. Elkady

Round-Robin Scheduling (RRS)

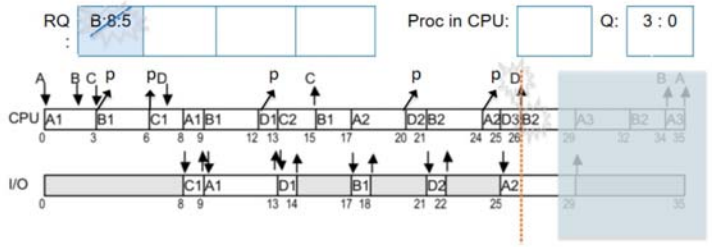
Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4 : 0	4	4 : 1	4	4 : 4
B	2	8 : 0	1	8 : 5	-	-
C	3	2 : 0	1	2 : 0	-	-
D	7	1 : 0	1	1 : 0	1	1 : 1



A. Elkady

Round-Robin Scheduling (RRS)

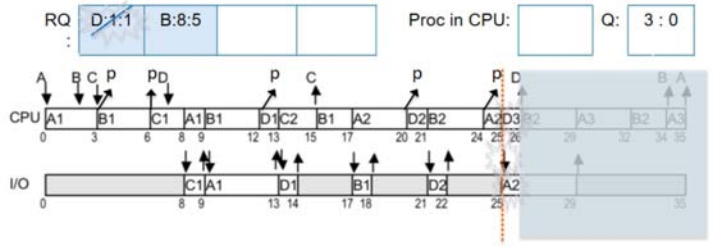
Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4 : 0	4	4 : 0	4	4 : 4
B	2	8 : 0	1	8 : 5	-	-
C	3	2 : 0	1	2 : 0	-	-
D	7	1 : 0	1	1 : 0	1	1 : 0



A. Elkady

Round-Robin Scheduling (RRS)

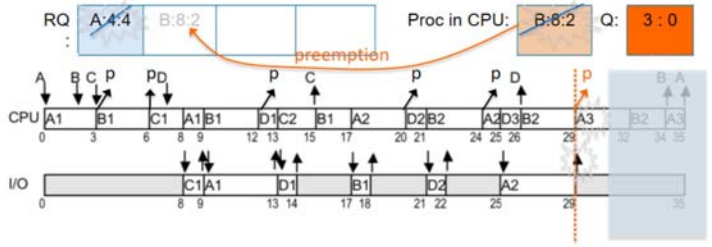
Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4 : 0	4	4 : 0	4	4 : 4
B	2	8 : 0	1	8 : 5	-	-
C	3	2 : 0	1	2 : 0	-	-
D	7	1 : 0	1	1 : 0	1	1 : 0



A. Elkady

Round-Robin Scheduling (RRS)

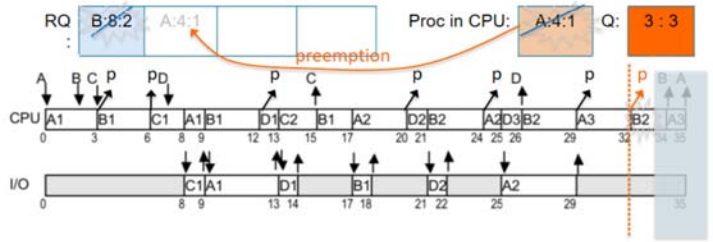
Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4 : 0	4	4 : 0	4	4 : 1
B	2	8 : 0	1	8 : 2	-	-
C	3	2 : 0	1	2 : 0	-	-
D	7	1 : 0	1	1 : 0	1	1 : 0



A. Elkady

Round-Robin Scheduling (RRS)

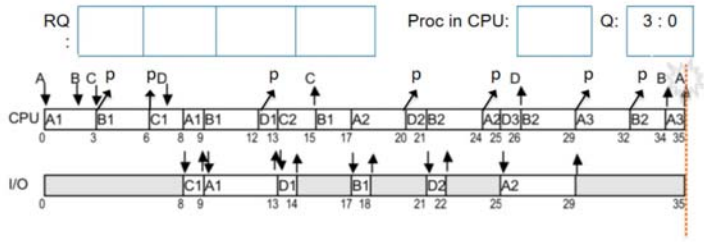
Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4 : 0	4	4 : 0	4	4 : 1
B	2	8 : 0	1	8 : 2	-	-
C	3	2 : 0	1	2 : 0	-	-
D	7	1 : 0	1	1 : 0	1	1 : 0



A. Elkady

Round-Robin Scheduling (RRS)

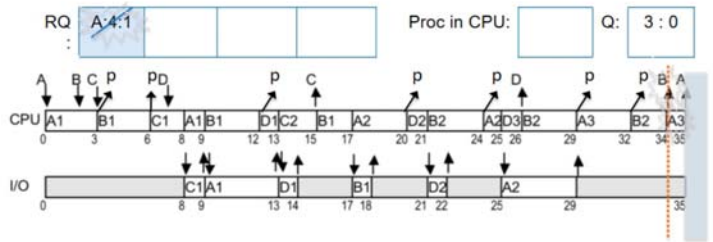
Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4 : 0	4	4 : 0	4	4 : 0
B	2	8 : 0	1	8 : 0	-	-
C	3	2 : 0	1	2 : 0	-	-
D	7	1 : 0	1	1 : 0	1	1 : 0



A. Elkady

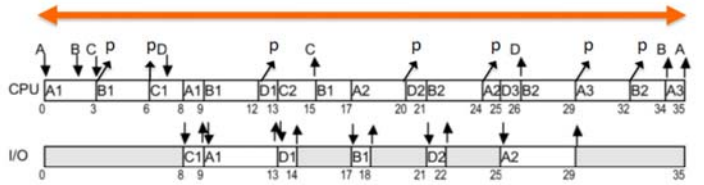
Round-Robin Scheduling (RRS)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4 : 0	4	4 : 0	4	4 : 1
B	2	8 : 0	1	8 : 0	-	-
C	3	2 : 0	1	2 : 0	-	-
D	7	1 : 0	1	1 : 0	1	1 : 0



A. Elkady

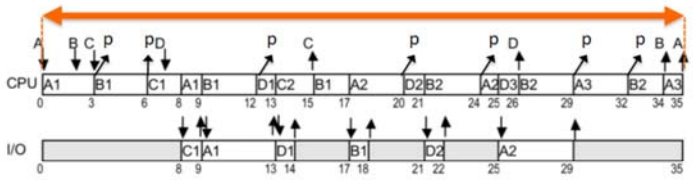
Round-Robin Scheduling (RRS)



- Processor utilization = (35 / 35) * 100 = 100 %
- Throughput = 4 / 35

A. Elkady

Round-Robin Scheduling (RRS)

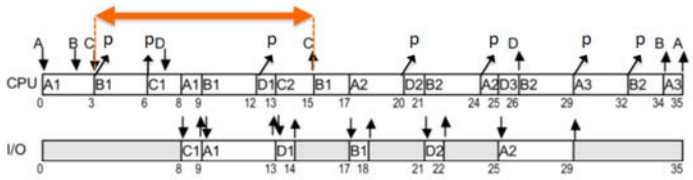


- Turn around time:
  - $tat_A = 35 - 0 = 35$
  - $tat_B = 34 - 2 = 32$
  - $tat_C = 15 - 3 = 12$
  - $tat_D = 26 - 7 = 19$

  $tat_{AVG} = (35 + 32 + 12 + 19) / 4 = 24.$

A. Elkady

Round-Robin Scheduling (RRS)

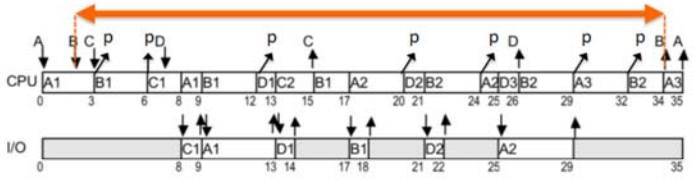


- Turn around time:
  - $tat_A = 35 - 0 = 35$
  - $tat_B = 34 - 2 = 32$
  - $tat_C = 15 - 3 = 12$
  - $tat_D = 26 - 7 = 19$

  $tat_{AVG} = (35 + 32 + 12 + 19) / 4 = 24.$

A. Elkady

Round-Robin Scheduling (RRS)

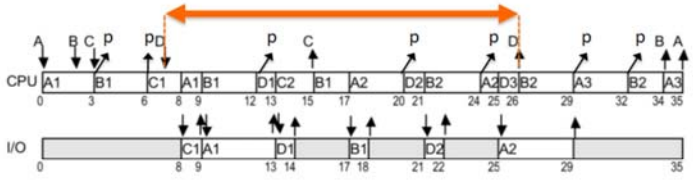


- Turn around time:
  - $tat_A = 35 - 0 = 35$
  - $tat_B = 34 - 2 = 32$
  - $tat_C = 15 - 3 = 12$
  - $tat_D = 26 - 7 = 19$

  $tat_{AVG} = (35 + 32 + 12 + 19) / 4 = 24.$

A. Elkady

Round-Robin Scheduling (RRS)

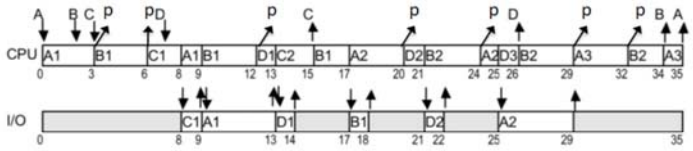


- Turn around time:
  - $tat_A = 35 - 0 = 35$
  - $tat_B = 34 - 2 = 32$
  - $tat_C = 15 - 3 = 12$
  - $tat_D = 26 - 7 = 19$

  $tat_{AVG} = (35 + 32 + 12 + 19) / 4 = 24.$

A. Elkady

Round-Robin Scheduling (RRS)

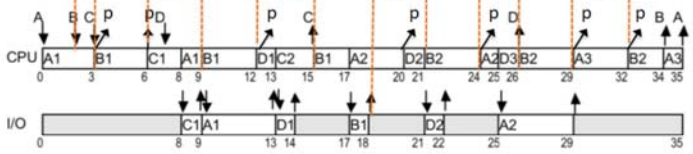


- Turn around time:
  - $tat_A = 35 - 0 = 35$
  - $tat_B = 34 - 2 = 32$
  - $tat_C = 15 - 3 = 12$
  - $tat_D = 26 - 7 = 19$

  $tat_{AVG} = (35 + 32 + 12 + 19) / 4 = 24$

A. Elkady

Round-Robin Scheduling (RRS)

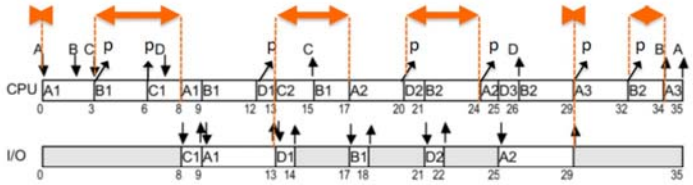


- Waiting time:
  - $wt_A = (0 - 0)+(8 - 3)+(17 - 13)+(24 - 20)+(29 - 29)+(34 - 32)=15$
  - $wt_B = (3 - 2)+(9 - 6)+ (15 -12)+(21 - 18)+(26 - 24)+(32 - 29) =15$
  - $wt_C = (6 - 3) + (13 - 9) = 7$
  - $wt_D = (12 - 7) + (20 - 14) + (25 - 22) = 14$

  $wt_{AVG} = (15 + 12 + 7 + 11) / 4 = 11.25$

A. Elkady

Round-Robin Scheduling (RRS)

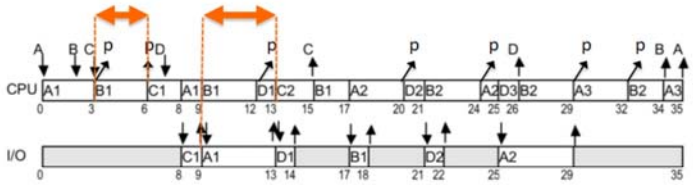


- Waiting time:
  - $wt_A = (0 - 0)+(8 - 3)+(17 - 13)+(24 - 20)+(29 - 29)+(34 - 32)=15$
  - $wt_B = (3 - 2)+(9 - 6)+ (15 -12)+(21 - 18)+(26 - 24)+(32 - 29) =15$
  - $wt_C = (6 - 3) + (13 - 9) = 7$
  - $wt_D = (12 - 7) + (20 - 14) + (25 - 22) = 14$

  $wt_{AVG} = (15 + 12 + 7 + 11) / 4 = 11.25$

A. Elkady

Round-Robin Scheduling (RRS)

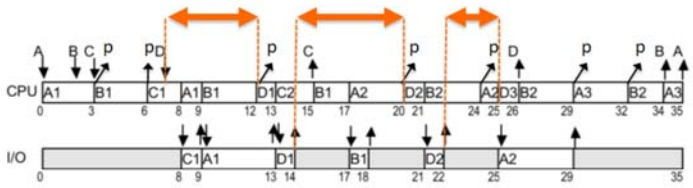


- Waiting time:
  - $wt_A = (0 - 0)+(8 - 3)+(17 - 13)+(24 - 20)+(29 - 29)+(34 - 32)=15$
  - $wt_B = (3 - 2)+(9 - 6)+ (15 -12)+(21 - 18)+(26 - 24)+(32 - 29) =15$
  - $wt_C = (6 - 3) + (13 - 9) = 7$
  - $wt_D = (12 - 7) + (20 - 14) + (25 - 22) = 14$

  $wt_{AVG} = (15 + 12 + 7 + 11) / 4 = 11.25$

A. Elkady

Round-Robin Scheduling (RRS)

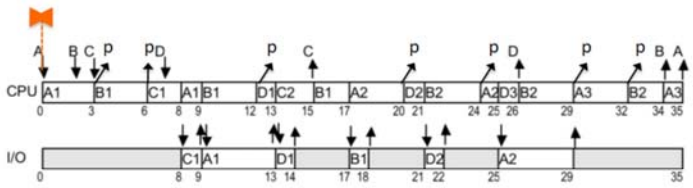


- Waiting time:
  - $wt_A = (0-0)+(8-3)+(17-13)+(24-20)+(29-29)+(34-32)=15$
  - $wt_B = (3-2)+(9-6)+(15-12)+(21-18)+(26-24)+(32-29)=15$
  - $wt_C = (6-3)+(13-9) = 7$
  - $wt_D = (12-7)+(20-14)+(25-22) = 14$

  $wt_{AVG} = (15 + 12 + 7 + 11) / 4 = 11.25$

A. Elkady

Round-Robin Scheduling (RRS)

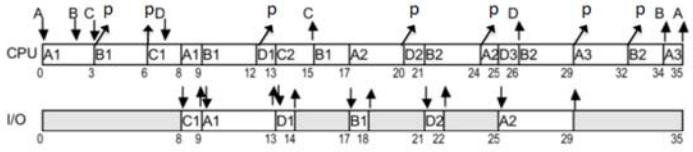


- Response time:
  - $rt_A = 0-0 = 0$
  - $rt_B = 36-2 = 1$
  - $rt_C = 6-3 = 3$
  - $rt_D = 12-7 = 5$

  $rt_{AVG} = (0 + 1 + 3 + 5) / 4 = 2.25$

A. Elkady

Round-Robin Scheduling (RRS)

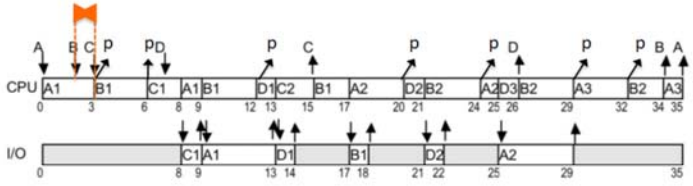


- Waiting time:
  - $wt_A = (0-0)+(8-3)+(17-13)+(24-20)+(29-29)+(34-32)=15$
  - $wt_B = (3-2)+(9-6)+(15-12)+(21-18)+(26-24)+(32-29)=15$
  - $wt_C = (6-3)+(13-9) = 7$
  - $wt_D = (12-7)+(20-14)+(25-22) = 14$

  $wt_{AVG} = (15 + 12 + 7 + 11) / 4 = 11.25$

A. Elkady

Round-Robin Scheduling (RRS)

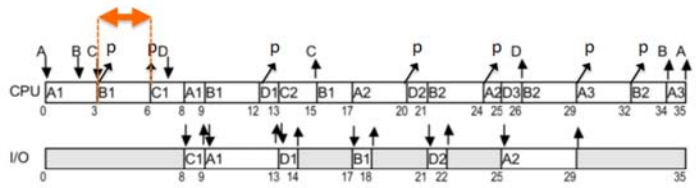


- Response time:
  - $rt_A = 0-0 = 0$
  - $rt_B = 3-2 = 1$
  - $rt_C = 6-3 = 3$
  - $rt_D = 12-7 = 5$

  $rt_{AVG} = (0 + 1 + 3 + 5) / 4 = 2.25$

A. Elkady

Round-Robin Scheduling (RRS)

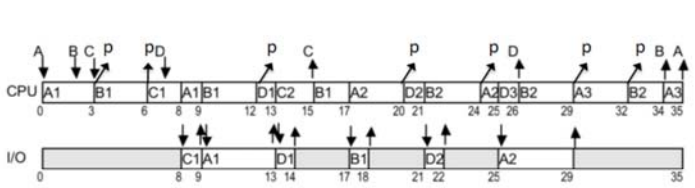


- Response time:
  - $rt_A = 0 - 0 = 0$
  - $rt_B = 36 - 2 = 1$
  - $rt_C = 6 - 3 = 3$
  - $rt_D = 12 - 7 = 5$

  $rt_{AVG} = (0 + 1 + 3 + 5) / 4 = 2.25$

A. Elkady

Round-Robin Scheduling (RRS)

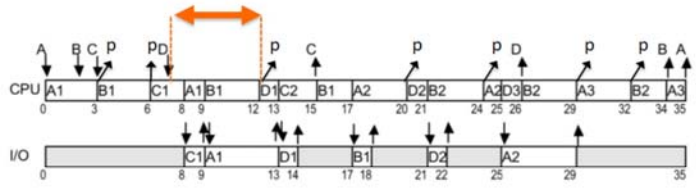


- Response time:
  - $rt_A = 0 - 0 = 0$
  - $rt_B = 3 - 2 = 1$
  - $rt_C = 6 - 3 = 3$
  - $rt_D = 12 - 7 = 5$

  $rt_{AVG} = (0 + 1 + 3 + 5) / 4 = 2.25$

A. Elkady

Round-Robin Scheduling (RRS)

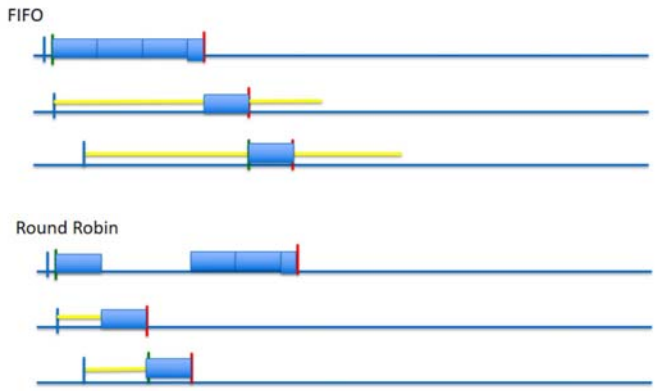


- Response time:
  - $rt_A = 0 - 0 = 0$
  - $rt_B = 36 - 2 = 1$
  - $rt_C = 6 - 3 = 3$
  - $rt_D = 12 - 7 = 5$

  $rt_{AVG} = (0 + 1 + 3 + 5) / 4 = 2.25$

A. Elkady

Round Robin vs FIFO



A. Elkady

Round-Robin Discussion

- How do you choose time slice?
  - What if too big?
    - Response time suffers
  - What if infinite (∞)?
    - Get back FCFS/FIFO
  - What if time slice too small?
    - Throughput suffers!

- Actual choices of timeslice:
  - Initially, UNIX timeslice one second:
    - Worked ok when UNIX was used by one or two people.
    - What if three compilations going on? 3 seconds to echo each keystroke!
  - In practice, need to balance short-job performance and long-job throughput:
    - Typical time slice today is between 10ms – 100ms
    - Typical context-switching overhead is 0.1ms – 1ms
    - Roughly 1% overhead due to context-switching

A. Elkady

More on Round Robin

- Round Robin assumes all processes are equally important
- Not true in real life
  - Interactive tasks need high priority for good response
  - We might want non-interactive tasks to get the CPU less frequently *(this goal led us to SRTF)*
  - Some tasks might be time critical
  - Users may have different status (e.g., administrator)
- Priority scheduling algorithm:
  - Each process has a priority number assigned to it
  - Pick the process with the highest priority
  - Processes with the same priority are scheduled round-robin

A. Elkady 298

comparison

	FCFS	SPF	SRTF	RR
tat_{avg}	28.25	23.50	18.75	24.50
wt_{avg}	16.50	10.50	6.25	12.25
rt_{avg}	4.50	3.00	1.25	2.25
	Easy to implement	Not possible to know next CPU burst exactly, it can only be guessed	Not possible to know next CPU burst exactly, it can only be guessed	Implementable, rt_{max} is important for interactive systems

A. Elkady

Interactive vs. batch scheduling

Batch

- First-Come-First-Served (FCFS)
- Shortest Job/Process First (SJF/SPF)
- Shortest Remaining Time First (SRTF)
- Priority (non-preemptive)

Interactive

- Round-Robin (RR) Priority (preemptive)
- Lottery

A. Elkady

Priority Scheduling

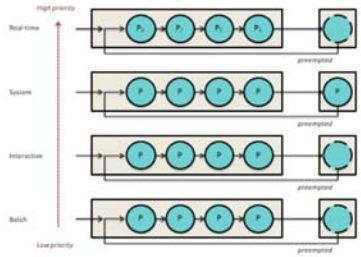
Round Robin assumes all processes are equally important

- Not true
  - Interactive tasks need high priority for good response
  - We might want non-interactive tasks to get the CPU less frequently:
    - this goal led us to SRTF
  - Some tasks might be time critical
  - Users may have different status (e.g., administrator)

Multi-Level Queue Scheduling

Does each task need to have a unique priority level?

- Priority classes: ready queues for each priority level
  - Each priority class gets its own queue
  - Processes are permanently assigned to a specific queue
  - Examples: *System processes, interactive processes, slow interactive processes, background non-interactive processes*
- Implementation:
  - Each queue may have a different scheduling algorithm
    - Usually round-robin
  - Assign a priority to each process
    - "Ready" process with highest priority allowed to run
    - Running process may be interrupted after its quantum expires



Priority Scheduling

- Assign a priority (number) to each process
- Schedule processes based on their priority
- CPU is given to the process with the highest priority
- *SPF is a special case of priority scheduling algorithm where priority is based on process time.*

Priority Scheduling – Problems

- Priority Inversion
  - A low-priority thread may not get scheduled, thereby preventing a high-priority thread that is waiting for a resource from making progress
- Starvation
  - A low priority thread may never get scheduled if there is always a high-priority thread ready to run
    - If the system is heavily loaded, it is a great probability that there is a higher-priority process to grab the processor.
- Solution?

Static & Dynamic Priorities

- Static priority
  - Priority never changes
  - Problem: Fixed priorities are too restrictive
- Dynamic Priorities
  - Priorities are altered over time, as process behavior changes!
  - One solution for the starvation problem might be to gradually increase the priority of processes that stay in the system for a long time.
- When do you change the priority of a process?
  - How? When? Why?

Multilevel Feedback Queues

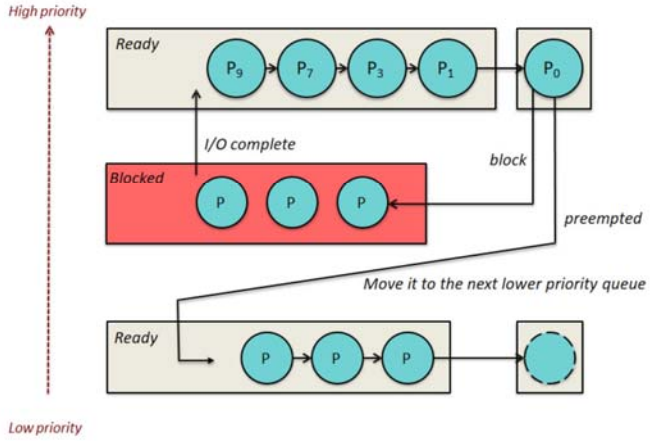
- *N* priority levels, round-robin scheduling within each level
- Goals
  - Allow processes to *move* between priority queues based on feedback
    - Have the scheduler learn the behavior of each task and adjust priorities
  - Separate processes based on CPU burst behavior
    - I/O-bound processes will end up on higher-priority queues
- Rules
  - A new process gets the highest priority
  - If a process does not finish its quantum (blocks on I/O)
    - Stay at the same priority level (round robin)
  - If process was preempted (time quantum expired)
    - Demote to the next lower priority level

Multi-Level Feedback Scheduling

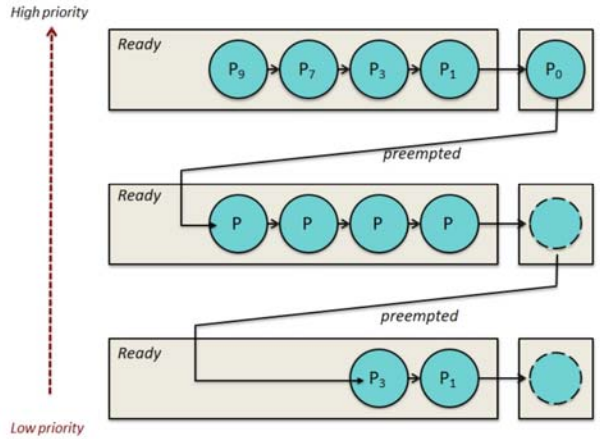
- Solution: Let the amount of CPU used be an indication of how a process is to be handled
  - Expired time quantum → more processing needed
  - Unexpired time quantum → less processing needed
- Adjusting quantum and frequency vs. adjusting priority?

Multilevel Feedback Queues

Pick the process from the head of the highest priority queue



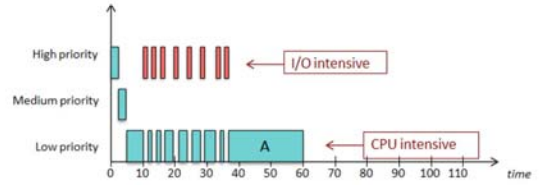
Multilevel Feedback Queues



A. Elkady

Multilevel Feedback Queues - Example

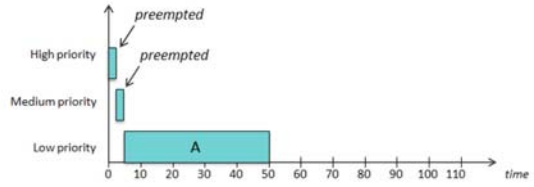
- Suppose a highly interactive process, *B* ( ), starts at *T=10*
- It never uses up its quantum
  - *B* gets priority but spends a lot of its time in the blocked state
  - A ( ) gets to run only when *B* is blocked



A. Elkady

Multilevel Feedback Queues - Example

One long-running process



A. Elkady

Multilevel Feedback Queues - Issues

- Two problems
  - Starvation:
    - If there are a lot of interactive processes, the CPU-bound processes will never get to run
  - Interactive process ending up at a low priority:
    - If a process was CPU intensive (e.g., initializing a game) but then became interactive, it is forever doomed to a low priority
- Solution: process aging
  - Increase the priority of a so it will be scheduled to run
    - Simplest approach: periodically, set all processes to the highest priority
  - If it remains CPU-intensive, its priority will quickly fall again

A. Elkady 315

Gaming the system

- Q: How to fool the MFQ scheduler to give my process more CPU time?
- What if you make an I/O operation just before the end of each time slice?
  - You get to stay at a high priority!
- Solution
  - Don't worry whether a process uses up its time slice
  - Instead, keep track of CPU time used over a larger time interval
    - If a process uses up its allotment, then lower its priority
    - Lower levels can have longer allotments

Multilevel Feedback Queues

- Advantage
  - Good for separating processes based on CPU burst needs
  - Give priority to I/O bound processes
  - No need to estimate interactivity! (Estimates were often flawed)
- Disadvantages
  - Priorities get controlled by the system.
    - A process is considered important because it uses a lot of I/O
  - Processes whose behavior changes may be poorly scheduled
  - System can be gamed by scheduling bogus I/O
    - … but we have workarounds

Varying time slices

Two thoughts

- Lower priority processes get longer time slices
  - Amortize the cost of context switching
  - CPU-intensive tasks don't get to run often. When they do, let them run for a longer time.
  - Interactive tasks rarely use up their quantum anyway. Keep it short

OR

- Higher priority processes get longer time slices
  - Measure CPU usage per process over a longer time interval
  - Low CPU users are interactive and get high priority
  - If an interactive process needs to do some computation for a while, let it do so at high priority

Interactive vs. batch scheduling

Batch

- First-Come-First-Served (FCFS)
- Shortest Job/Process First (SJF/SPF)
- Shortest Remaining Time First (SRTF)
- Priority (non-preemptive)

Interactive

- Round-Robin (RR) Priority (preemptive)
- Lottery

Lottery scheduling

- Give every job some number of "lottery tickets" for CPU time
  - More tickets => higher share of CPU.
  - **To approximate SRTF,** short running jobs get more, long running jobs get fewer
  - **To avoid starvation**, every job gets at least one ticket
- On each time slice, randomly pick a winning ticket.
  - If there are n tickets, pick a number from 1 to n
  - Process holding the ticket gets to run for a quantum

- On average, CPU time is proportional to the number of tickets given to each job.
- **Advantage over strict priority scheduling:** behaves gracefully as load changes
  - Adding or deleting a job affects all jobs proportionally, independent of how many tickets each job possesses
- Over the long run, each process gets the CPU m/n of the time if the process has m of the n existing tickets
- Tickets can be transferred
  - Cooperating processes can exchange tickets
  - Clients can transfer tickets to server so it can have a higher priority

Next Time

Concurrency & Other Issues

Quiz

- What are the main tasks of the scheduler?
- What is the difference between preemptive and non-preemptive scheduling?
- What is the advantage of a shorter scheduling quantum?
- What is the advantage of a longer scheduling quantum?
- Why is feedback scheduling useful for interactive jobs?
- Are these scheduling policies subject to starvation?
  1. Shortest Job First scheduling?
  2. Round Robin scheduling?
  3. First Come First Served scheduling?
  4. Priority scheduling?
  5. SRTF?

Illustration: "Too much milk"

Went to buy milk

Time	Person A	Person B
3:00	Look in Fridge. Out of milk	
3:05	Leave for store	
3:10	Arrive at store	Look in Fridge. Out of milk
3:15	Buy milk	Leave for store
3:20	Arrive home, put milk away	Arrive at store
3:25		Buy milk
3:30		Arrive home, put milk away …