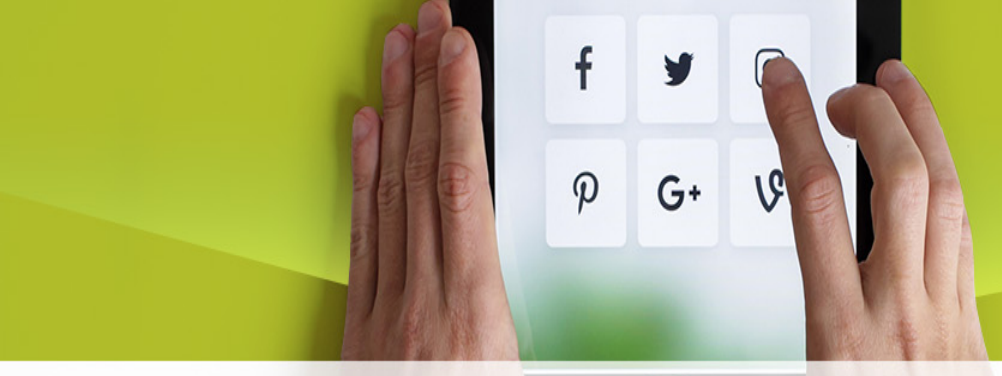
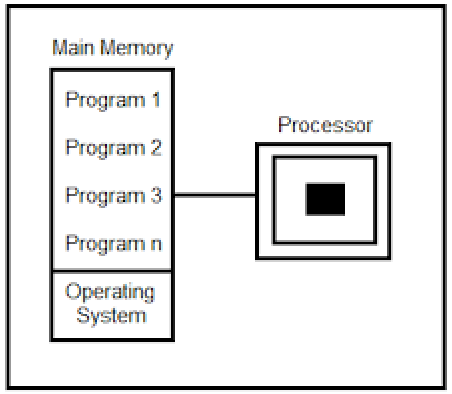
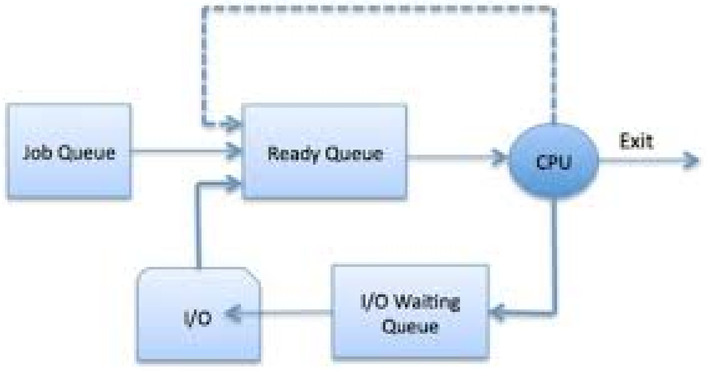
Operating Systems

Process Scheduling and Switching



Introduction



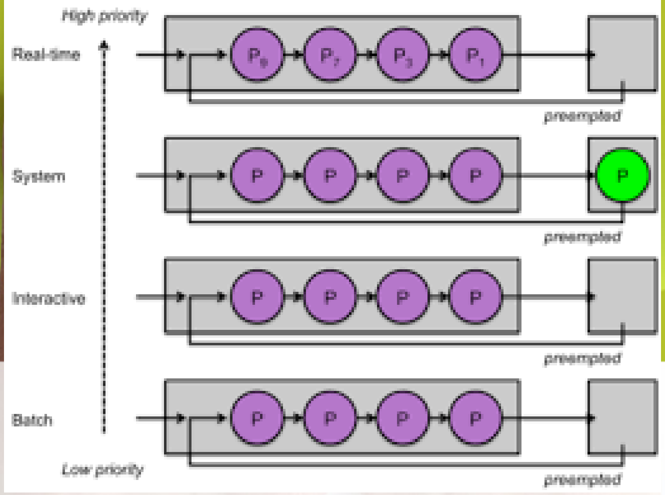


- An important aspect of multiprogramming is scheduling.

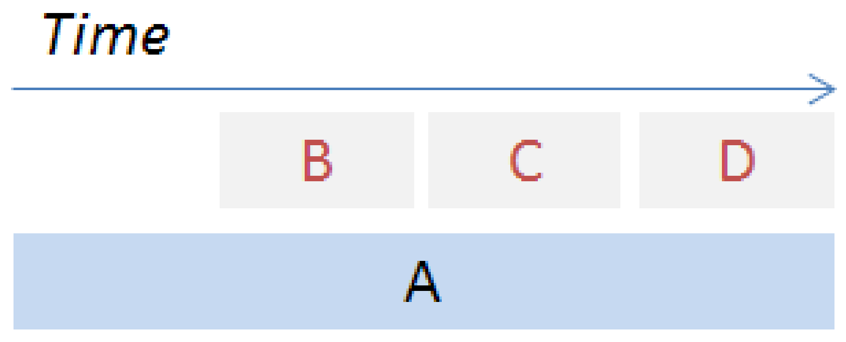
The resources that are scheduled are IO and processors.

- The goal is to achieve
  - High processor utilization
  - High throughput
    - number of processes completed per unit time
  - Low response time
    - time elapse from the submission of a request to the beginning of the response

Processor Scheduling

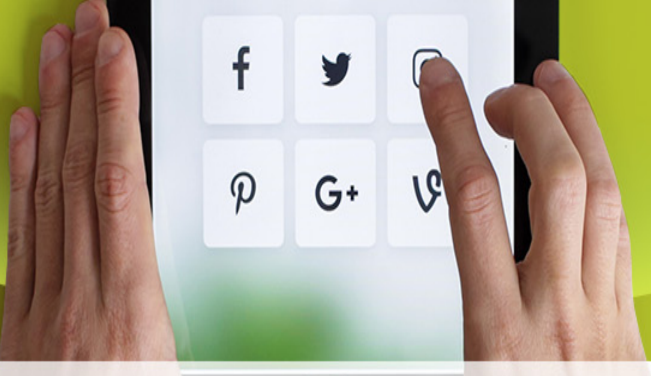


- Maximize CPU use, quickly switch processes onto CPU for time sharing.
- Process scheduler selects among available processes for next execution on CPU.
- Maintains scheduling queues of processes:
  - Job queue – set of all processes in the system.
  - Ready queue – set of all processes residing in main memory, ready and waiting to execute.
  - Device queues – set of processes waiting for an I/O device.
- Processes migrate among the various queues.



Scheduling Criteria

- CPU utilisation: During heavy loads, the CPU is busy almost 90% and in the lighter loads it is only active around 40%
- Throughput: the total number of processes that gets completed in unit of time is called throughput.
- Turnaround time: the time span from submission of a process to the system until is completed
- Waiting Time: the time spent by a process in a different queues
- Response Time: the time taken by a process in producing its first response after submission.



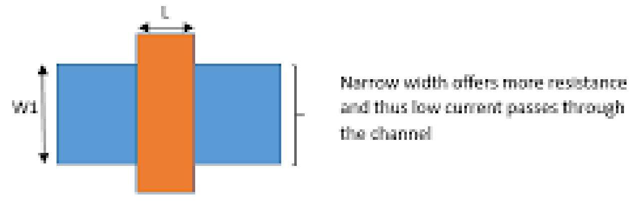
Requirements of scheduling

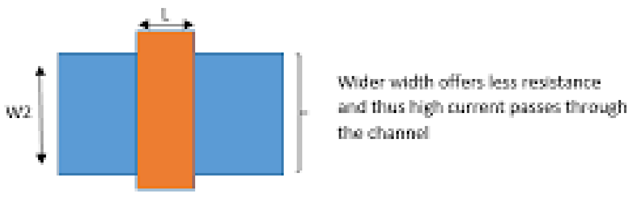
- An ideal scheduling discipline
  - is easy to implement
  - is fair and protective
  - provides performance bounds
- Each scheduling discipline makes a different trade-off among these requirements

Ease of implementation

- Scheduling discipline has to make a decision once every few microseconds!
- Should be implementable in a few instructions or hardware
  - for hardware: critical constraint is VLSI *space*
  - *Complexity of enqueue + dequeue processes*

Work per packet should scale less than linearly with number of active connections


L
W1
Narrow width offers more resistance and thus low current passes through the channel

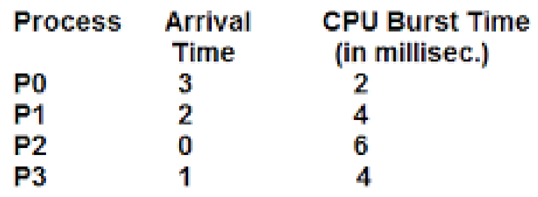

L
W2
Wider width offers less resistance and thus high current passes through the channel

Types of Scheduling

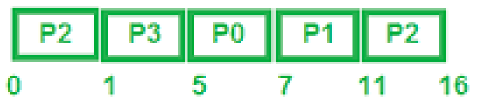
- Preemptive
- Non-Preemptive

Preemptive Scheduling

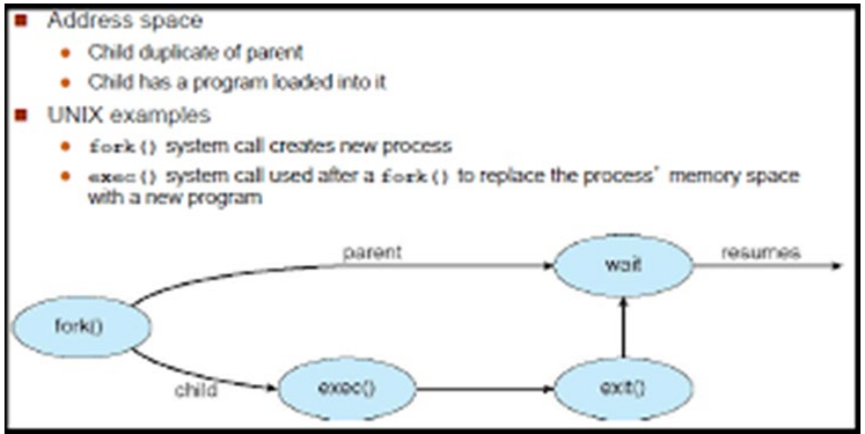
- When the CPU switches from one process to another before its completion, then is called preemptive scheduling
- Reasons why CPU leaves one process:
  - Some higher priority process arrives in the system
  - An interrupt occurs in a process
  - A child process comes into a parent process



Process	Arrival Time	CPU Burst Time (in millisec.)
P0	3	2
P1	2	4
P2	0	6
P3	1	4



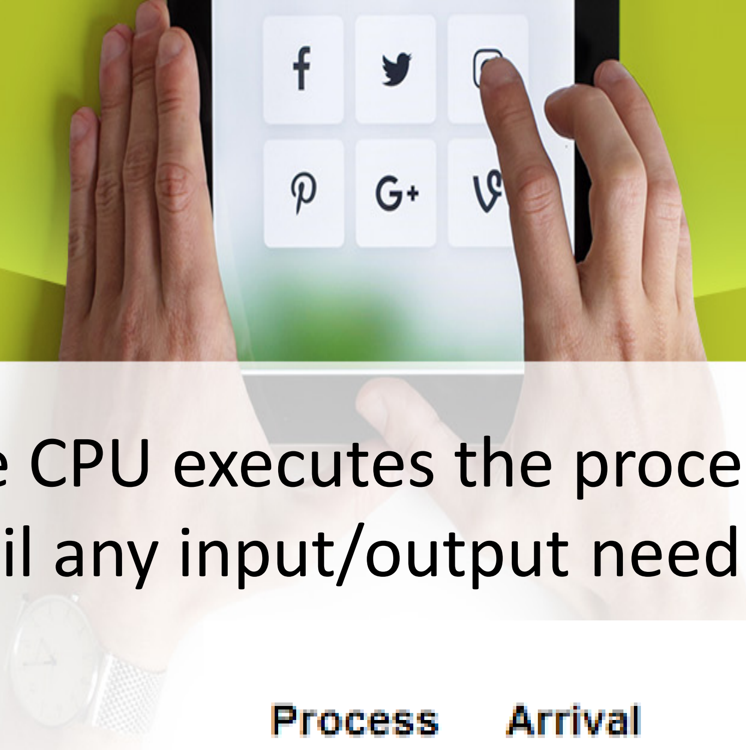
Preemptive Scheduling



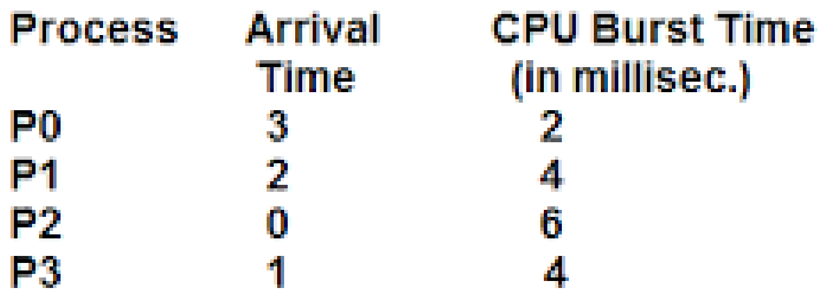
- Address space
  - Child duplicate of parent
  - Child has a program loaded into it
- UNIX examples
  - `fork()` system call creates new process
  - `exec()` system call used after a `fork()` to replace the process' memory space with a new program

parent
wait
resumes
fork()
child
exec()
exit()

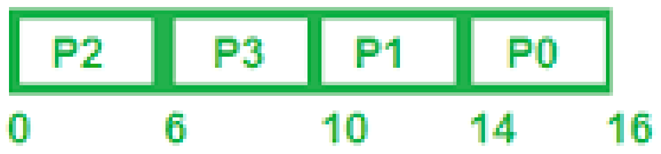
Non-preemptive Scheduling



- The CPU executes the process until it is terminated or until any input/output need arise.



Process	Arrival Time	CPU Burst Time (in millisec.)
P0	3	2
P1	2	4
P2	0	6
P3	1	4



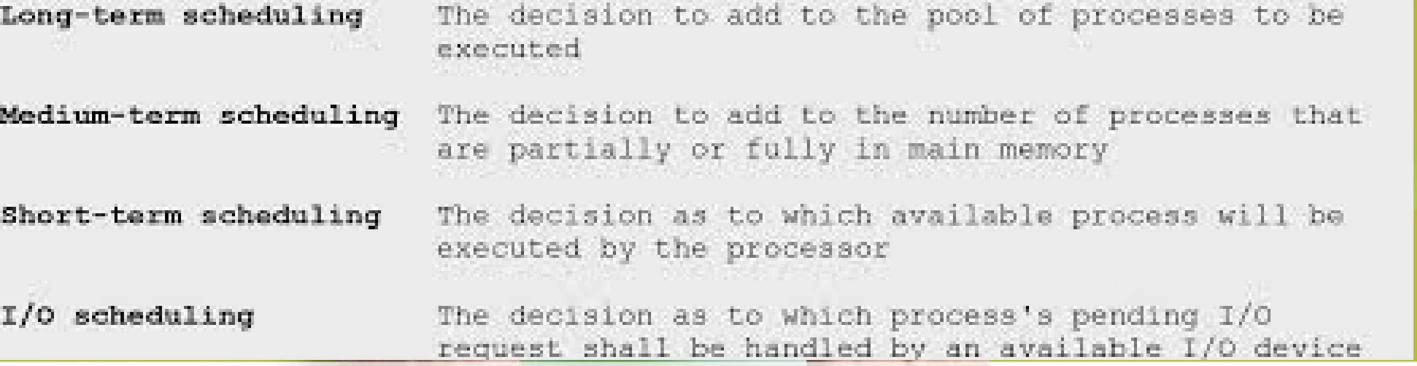
Non-Preemtive Scheduling

Self-Quiz

– Define CPU Scheduling
– Define turnarounds time

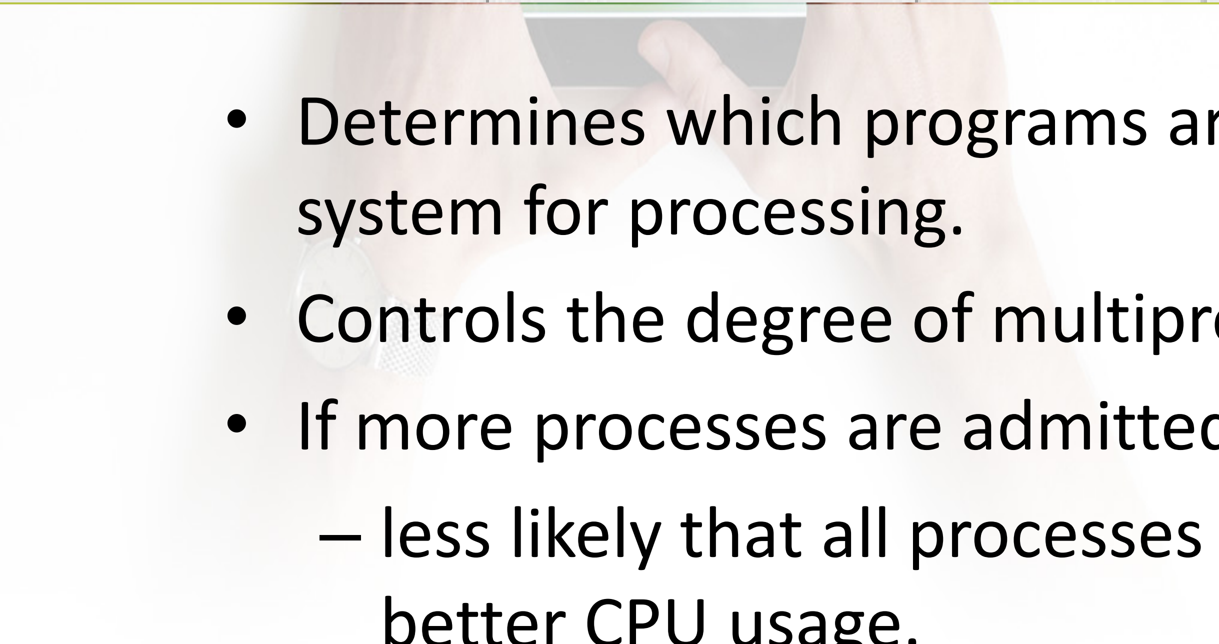
Types of Schedulers

1. Long-term scheduler (jobs scheduler) – selects which programs/processes should be brought into the ready queue.
2. Medium-term scheduler (emergency scheduler) – selects which job/process should be swapped out if system is loaded.
3. Short-term scheduler (CPU scheduler) – selects which process should be executed next and allocates CPU.



Long-term scheduling	The decision to add to the pool of processes to be executed
Medium-term scheduling	The decision to add to the number of processes that are partially or fully in main memory
Short-term scheduling	The decision as to which available process will be executed by the processor
I/O scheduling	The decision as to which process's pending I/O request shall be handled by an available I/O device

Long-Term Scheduling

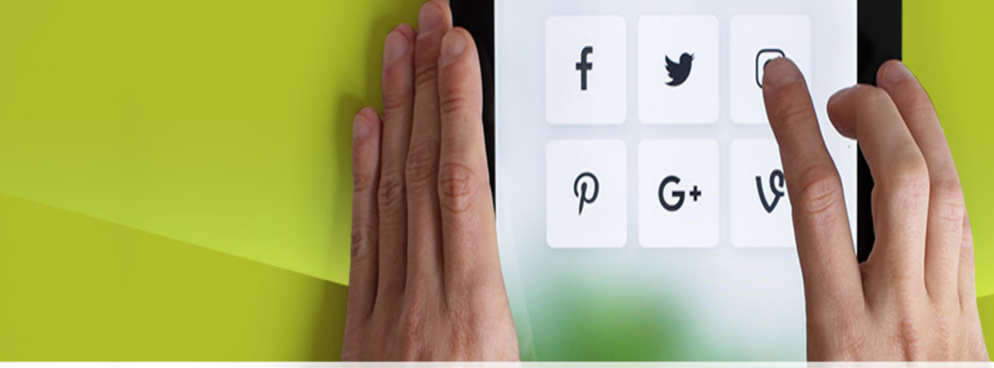


- Determines which programs are admitted to the system for processing.
- Controls the degree of multiprogramming.
- If more processes are admitted:
  - less likely that all processes will be blocked – better CPU usage.
  - each process has less fraction of the CPU.
- Long-term scheduler strives for good process mix.

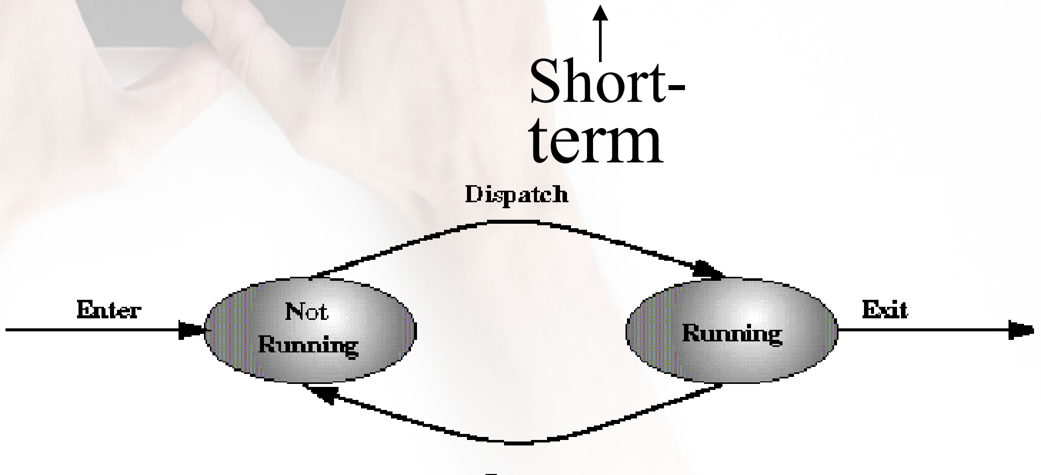
Short-Term Scheduling

- Determines which process is going to execute next (also called CPU scheduling).
- The short term scheduler is also known as the dispatcher (which is part of it).
- Is invoked on a event that may lead to choose another process for execution:
  – clock interrupts
  – I/O interrupts
  – operating system calls and traps
  – signals

Long/Short-Term Scheduling

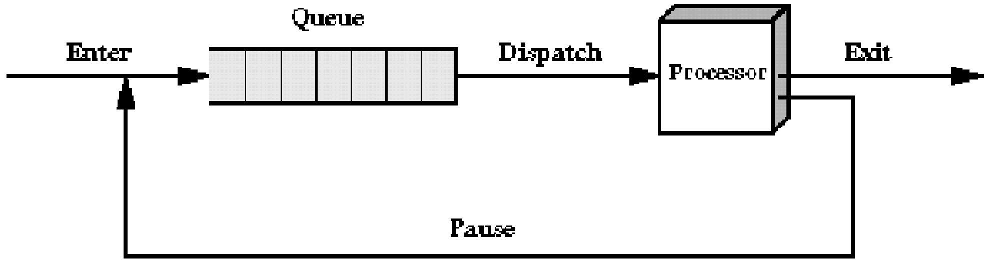


Short-term



(a) State transition diagram

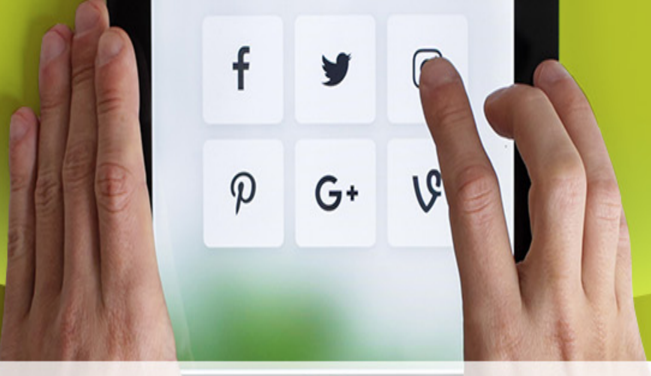
Long-term



(b) Queuing diagram

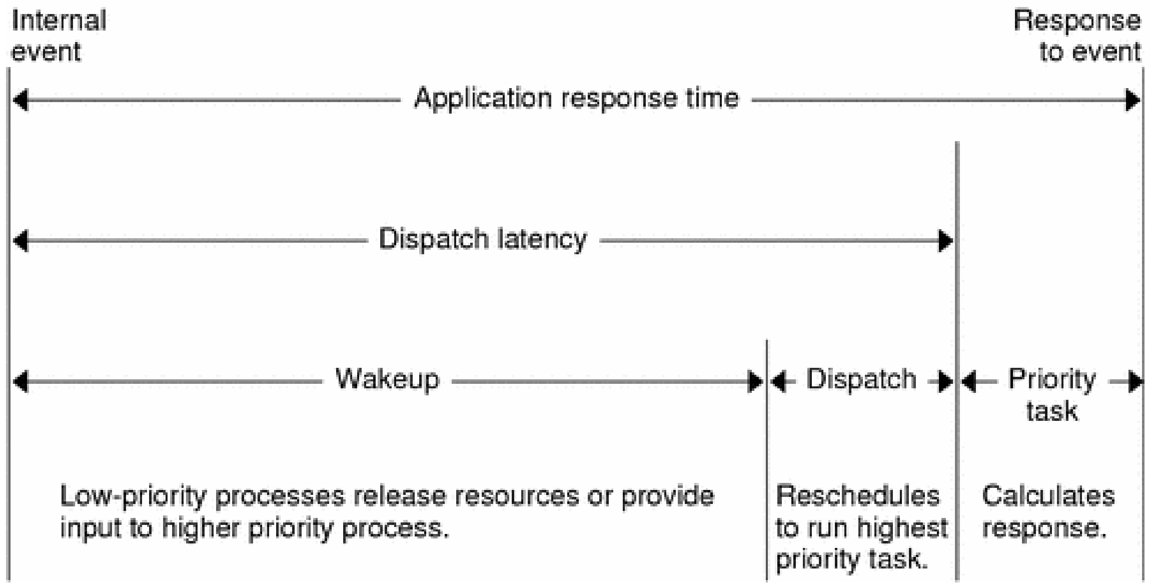
Dispatcher (short-term scheduler)

- Is an OS program that moves the processor from one process to another.
- It prevents a single process from monopolizing processor time.
- It decides who goes next according to a scheduling algorithm.
- The CPU will always execute instructions from the dispatcher while switching from process A to process B.



Dispatcher

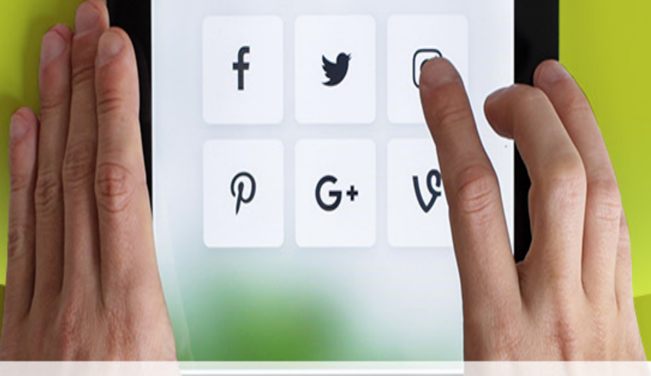
- Is a module of a OS that provides control of the CPU to the process which is selected by the short time scheduler.
- The dispatcher should be as fast as possible.
- The time consumed by the dispatcher is known as dispatch latency



Internal event
Response to event
Application response time
Dispatch latency
Wakeup
Dispatch
Priority task
Low-priority processes release resources or provide input to higher priority process.
Reschedules to run highest priority task.
Calculates response.

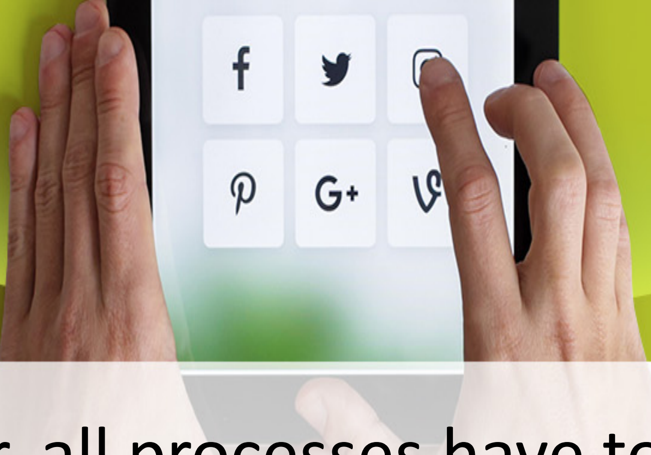
Degree of multiprogramming.

- The **degree of multiprogramming** describes the maximum number of processes that a single-processor system can accommodate efficiently.
- The primary factor affecting the **degree of multiprogramming** is the amount of memory available to be allocated to executing processes.



Aspects of Schedulers

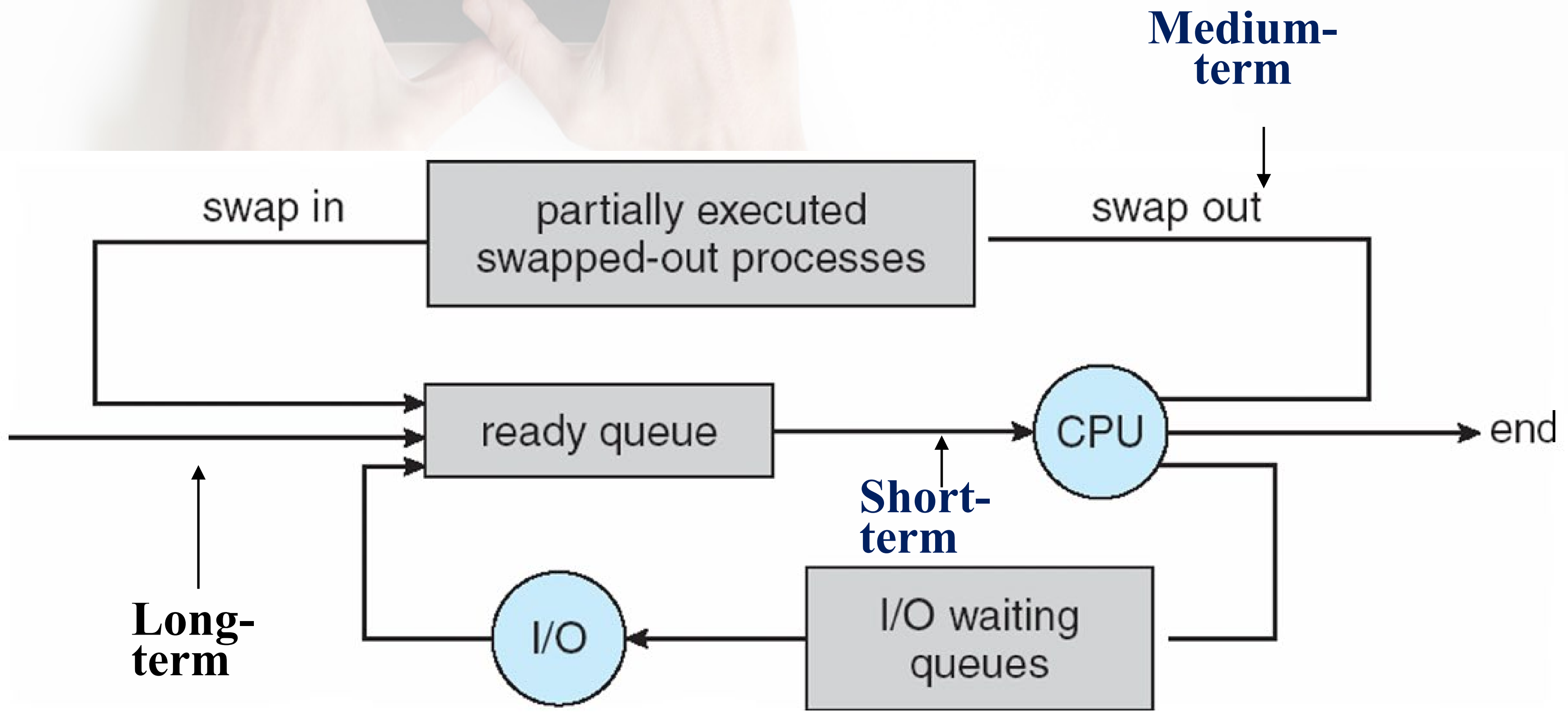
- Long-term scheduler is invoked very infrequently (seconds, minutes) ⇒ (may be slow).
- The long-term scheduler controls the degree of multiprogramming.
- Short-term scheduler is invoked very frequently (milliseconds) ⇒ (must be fast).
- Processes can be described as either:
  - **I/O-bound process** – spends more time doing I/O than computations, many short CPU bursts.
  - **CPU-bound process** – spends more time doing computations; few very long CPU bursts.

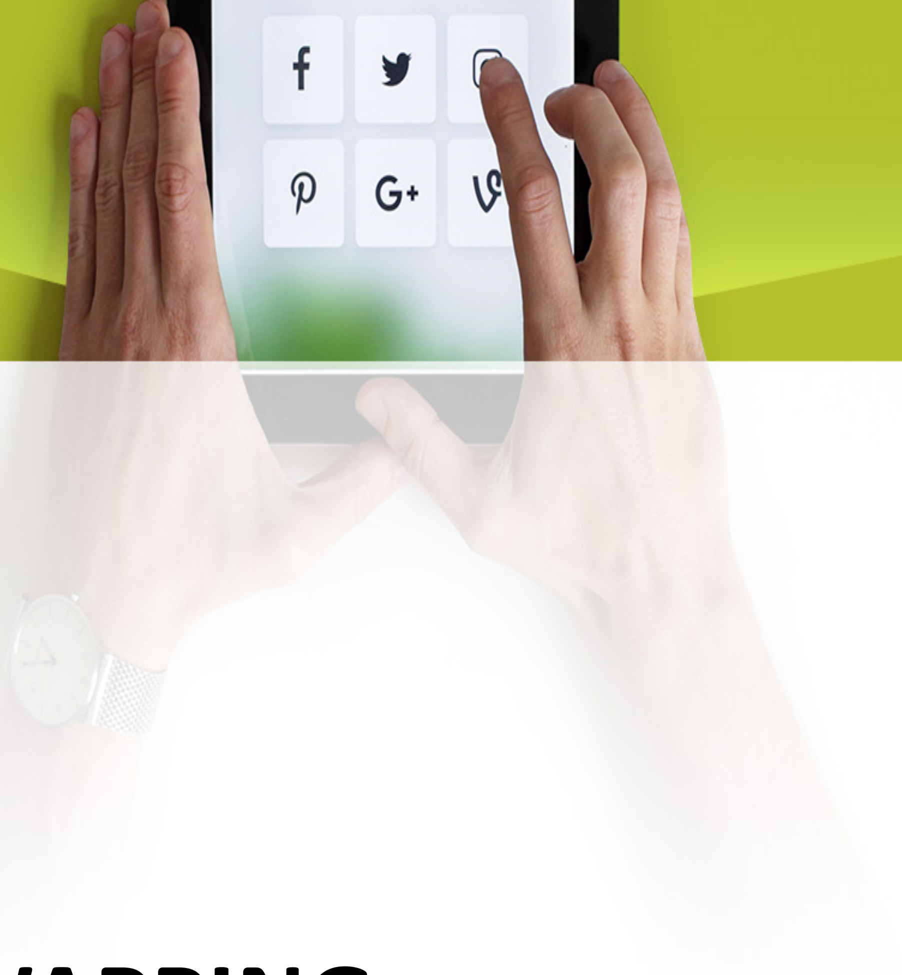


Medium-Term Scheduling

- So far, all processes have to be (at least partly) in main memory.
- Even with virtual memory, keeping too many processes in main memory will deteriorate the system’s performance.
- The OS may need to swap out some processes to disk, and then later swap them back in.
- Swapping decisions based on the need to manage multiprogramming.

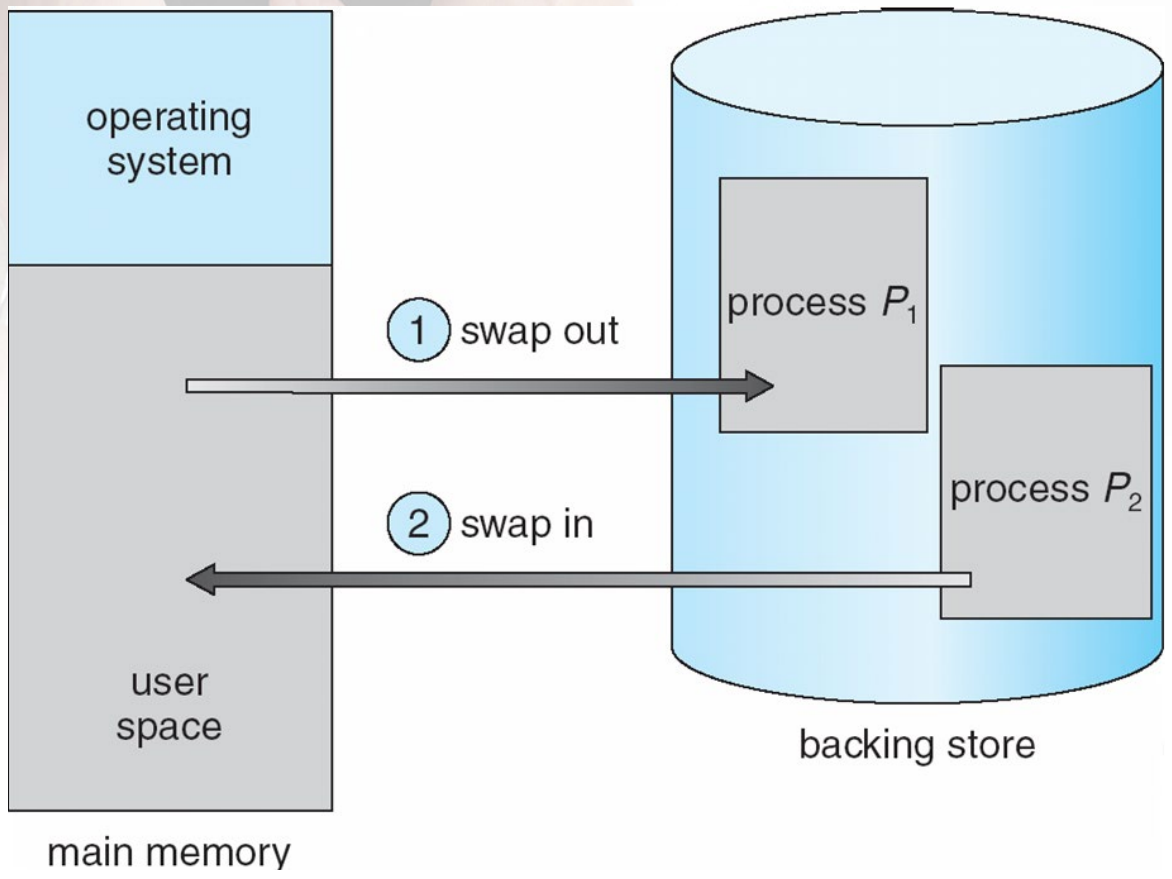
Addition of Medium Term Scheduling





SWAPPING

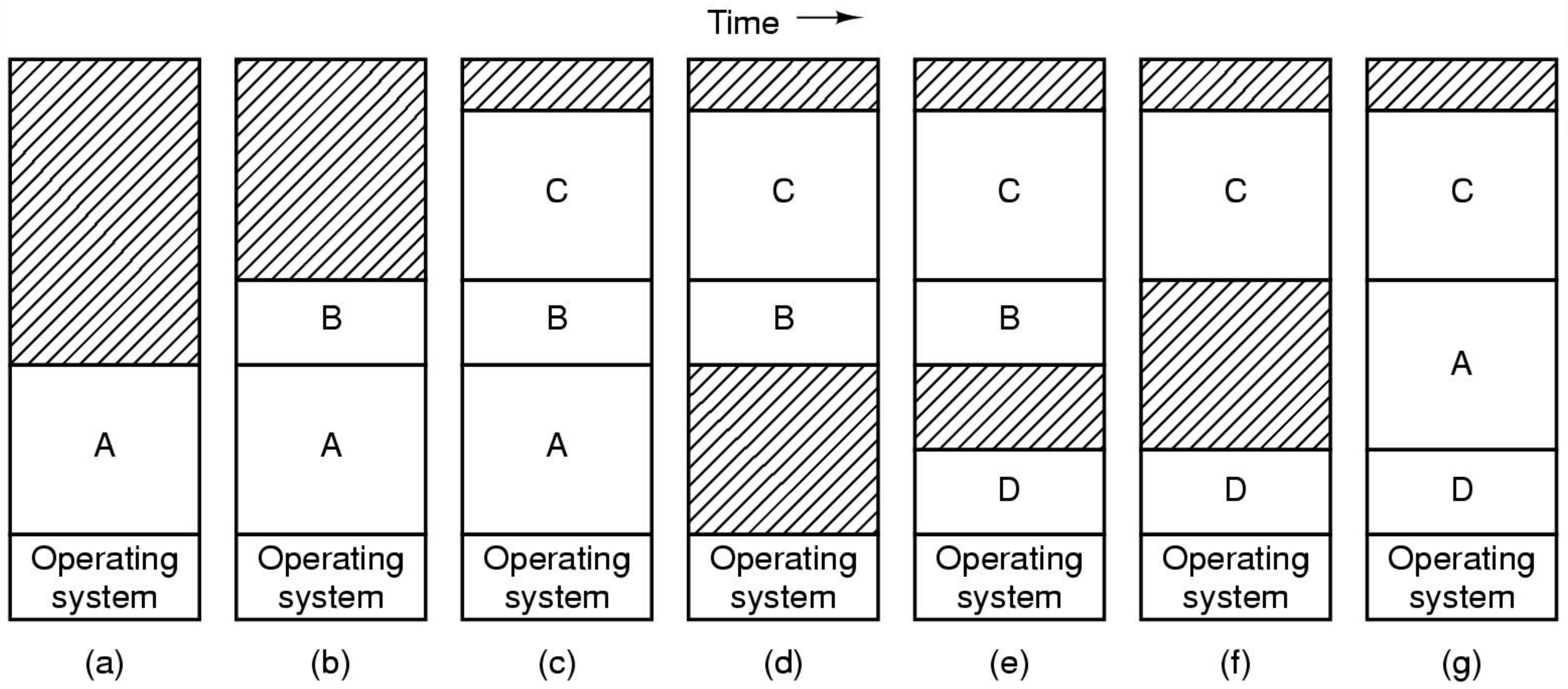
Schematic View of Swapping



Dynamics of Swapping

- A process can be swapped temporarily out of memory to a backing store, and then brought back into memory for continued execution
- **Backing store** – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images.
- **Roll out, roll in** – swapping variant used for priority-based scheduling algorithms; lower-priority process is swapped out so higher-priority process can be loaded and executed.
- Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped.
- Modified versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows).
- System maintains a ready queue of ready-to-run processes which have memory images on disk

Swapping Example



Support for Swapping

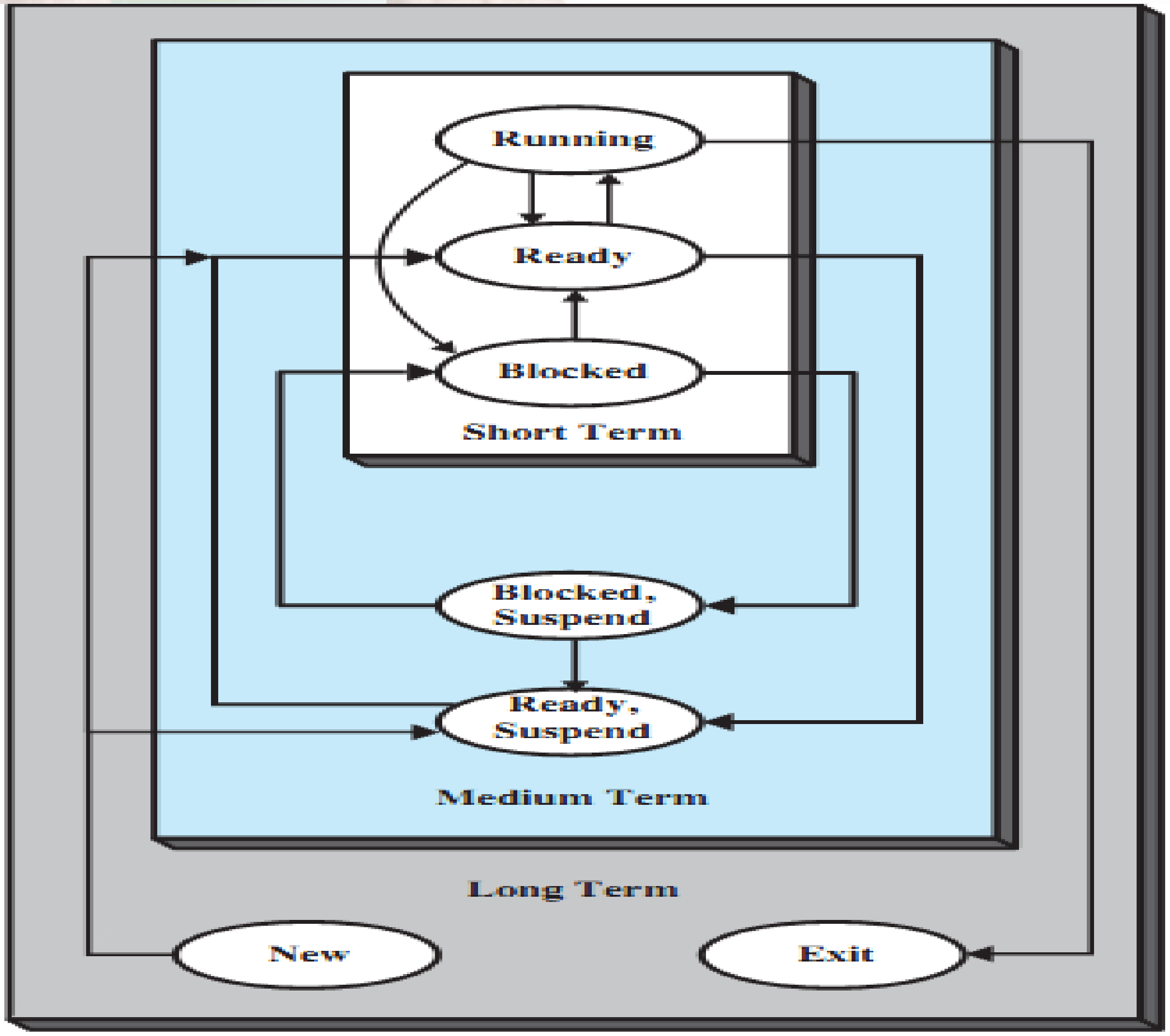
- The OS may need to suspend some processes, i.e., to swap them out to disk and then swap them back in.
- We add 2 new states:
  - Blocked Suspend: blocked processes which have been swapped out to disk.
  - Ready Suspend: ready processes which have been swapped out to disk.

STATE TRANSITIONS

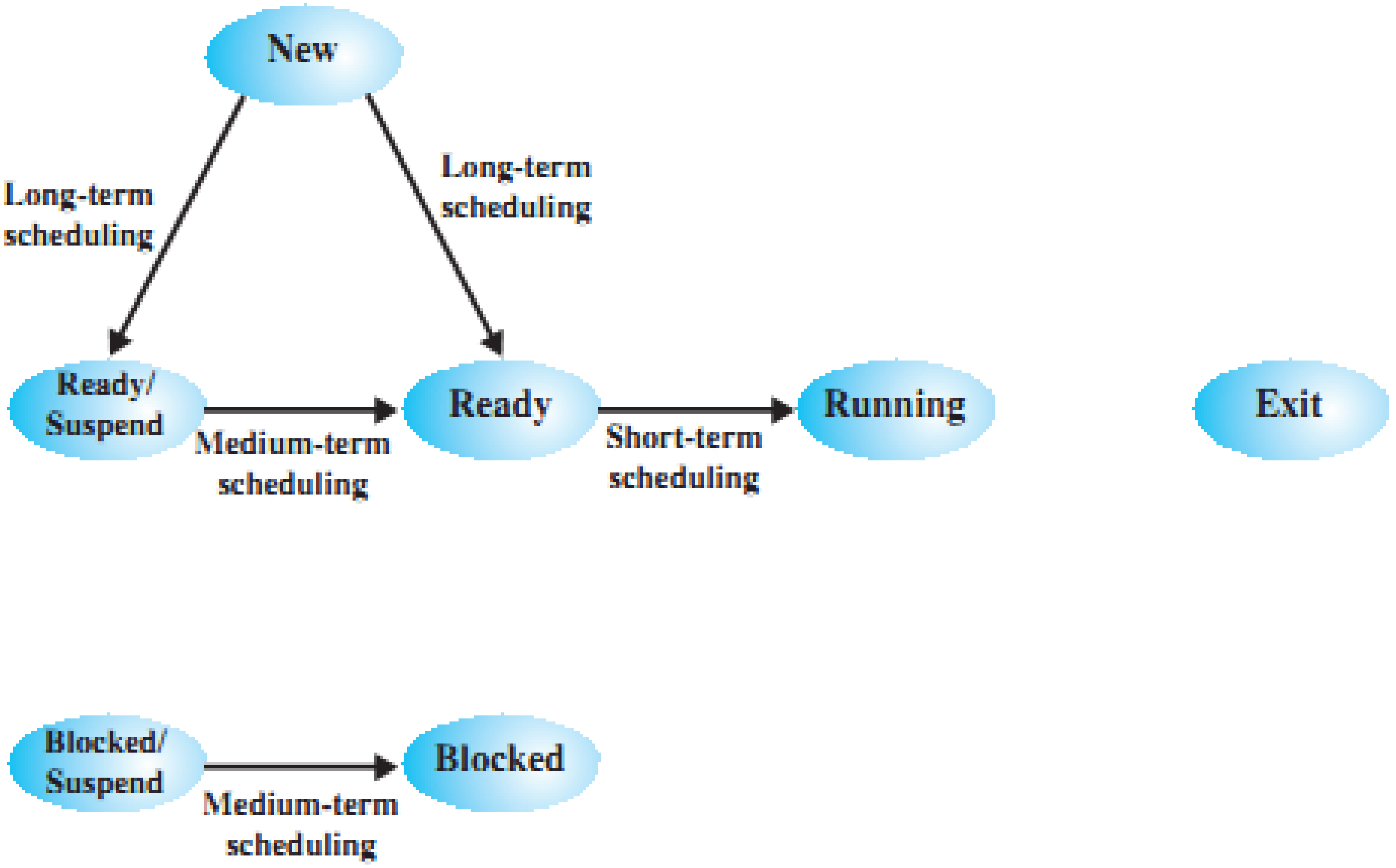
New state transitions

- Blocked —> Blocked Suspend
  - When all processes are blocked, the OS will make room to bring a ready process in memory.
- Blocked Suspend —> Ready Suspend
  - When the event for which it has been waiting occurs (state info is available to OS).
- Ready Suspend —> Ready
  - when no more ready processes in main memory.
- Ready —> Ready Suspend (unlikely)
  - When there are no blocked processes and must free memory for adequate performance.

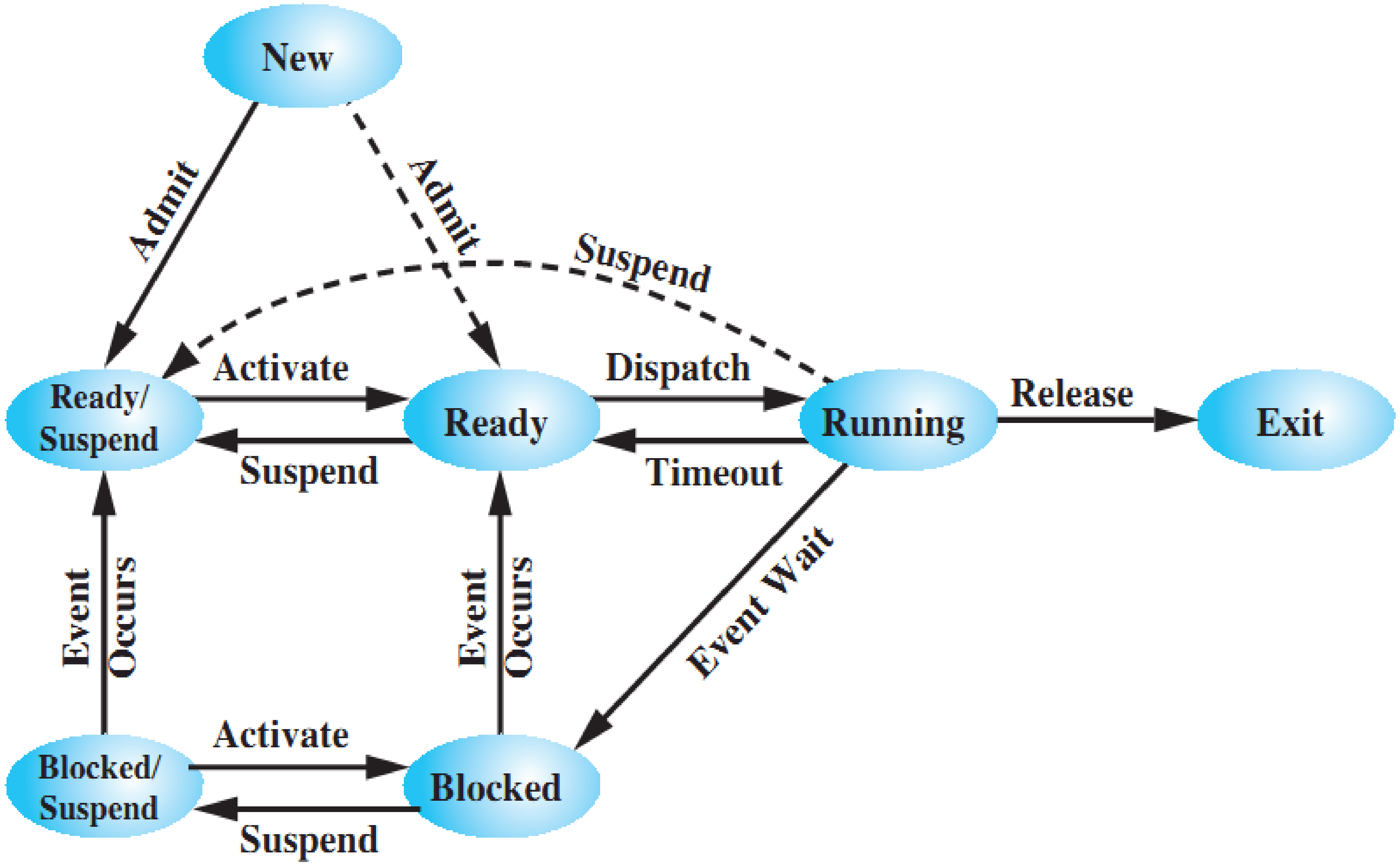
Another view of the 3 levels of scheduling

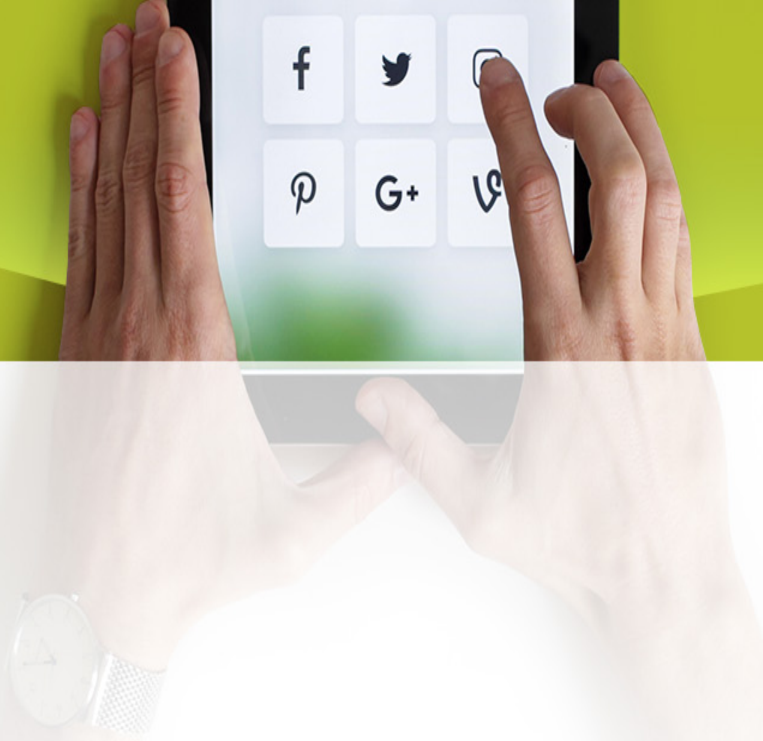


Classification of Scheduling Activity



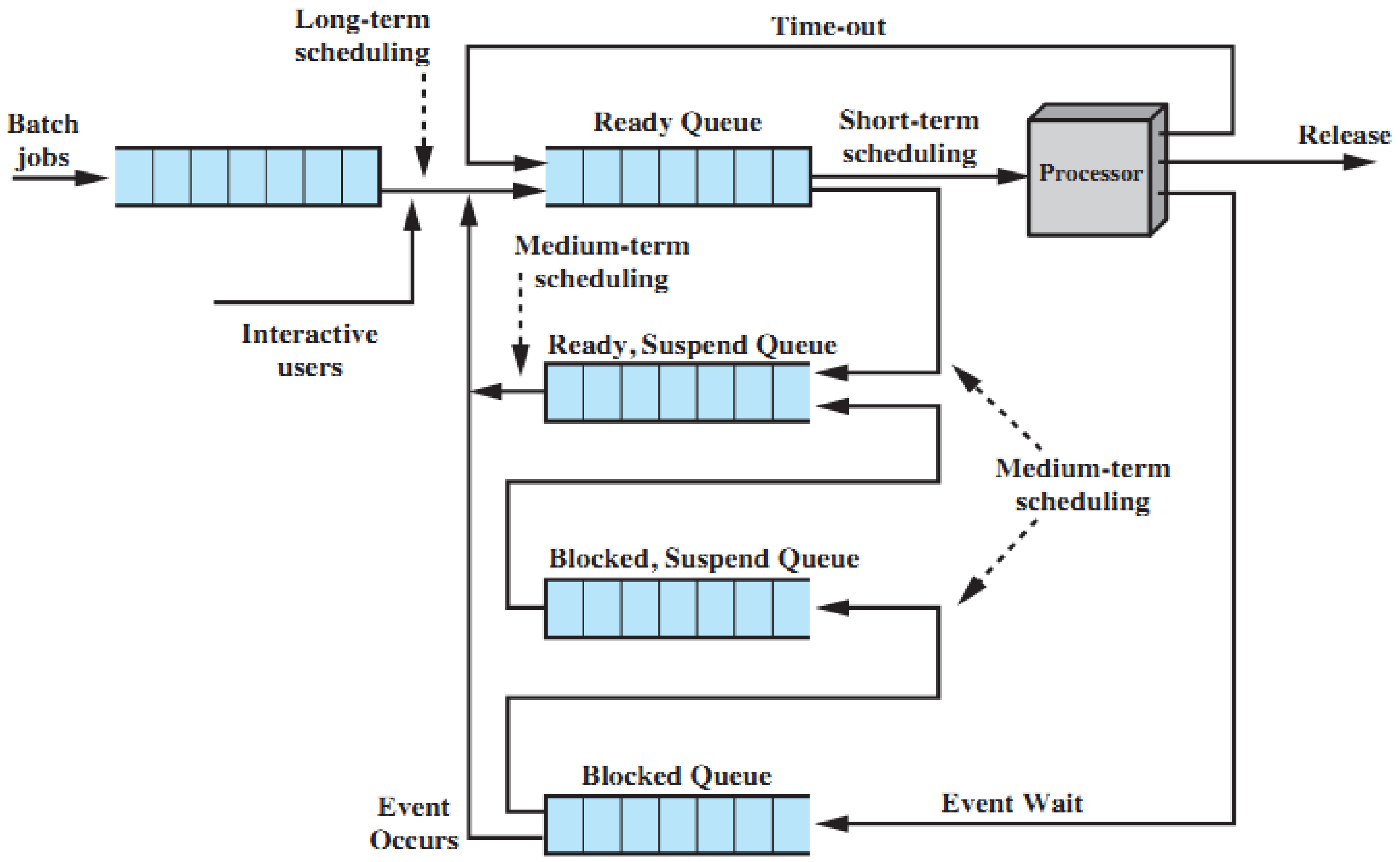
A Seven-state Process Model





QUEUING

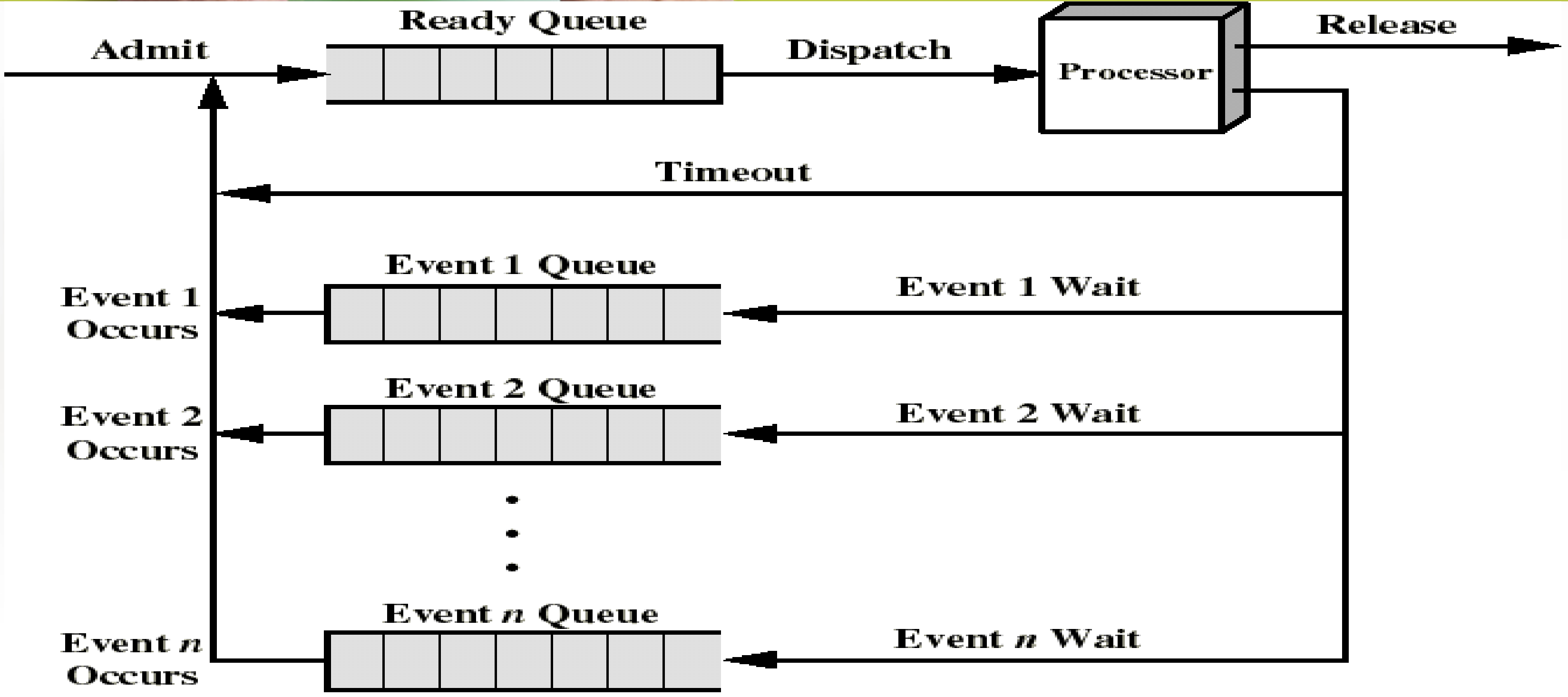
Queuing Diagram for Scheduling



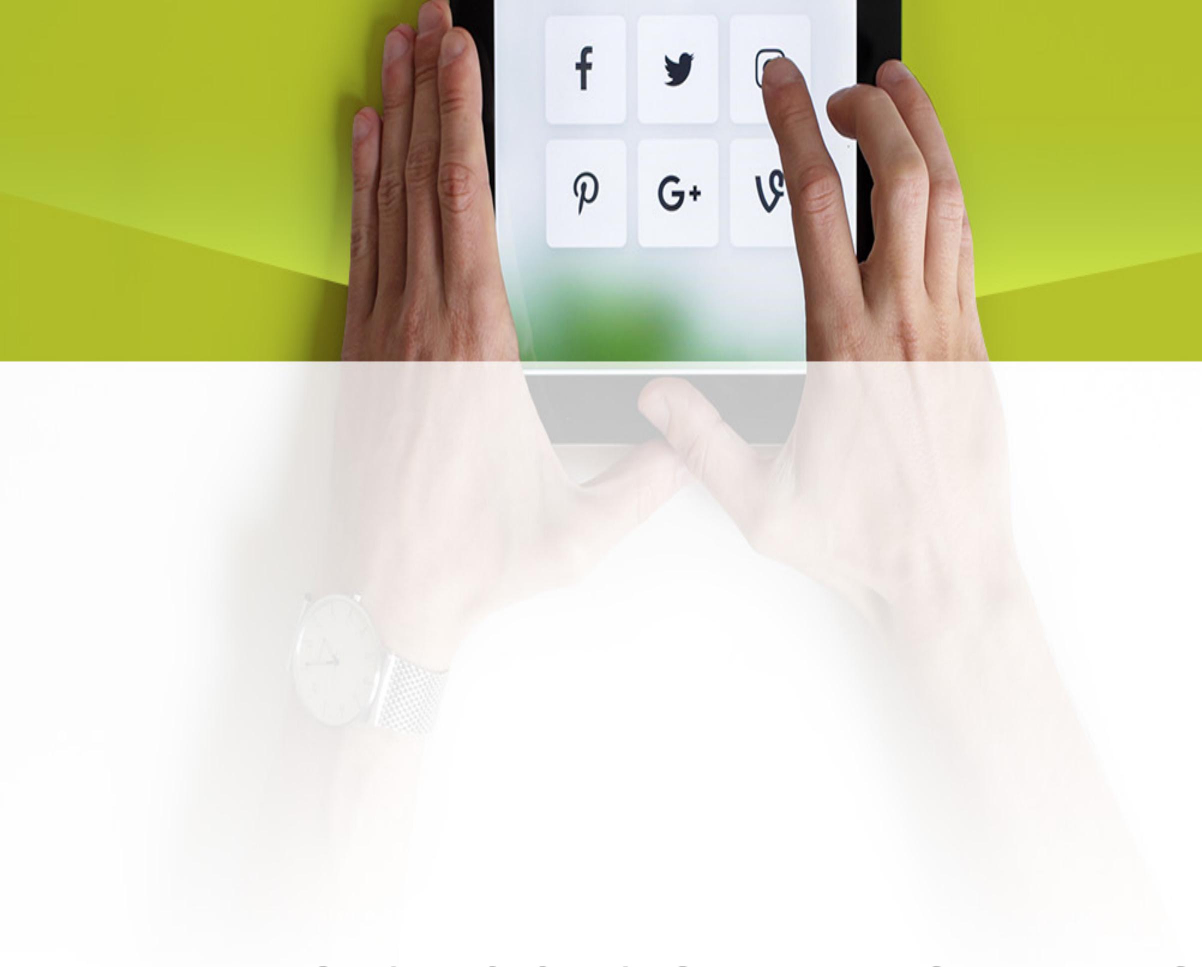
Process Scheduling Queues

- Process queue – set of *all* processes in the system.
- Ready queue – set of processes residing in main memory, ready and waiting to execute.
- Device queues – set of processes waiting for an I/O device.
- Processes migrate among the various queues.

A Queuing Discipline

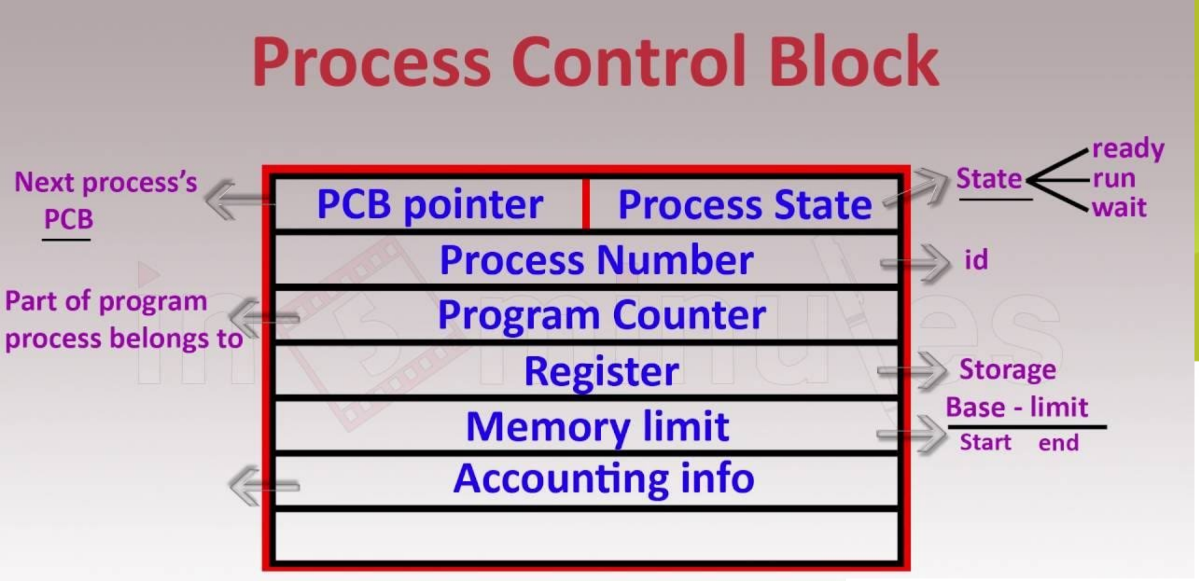


- **When event n occurs, the corresponding process is moved into the ready queue**

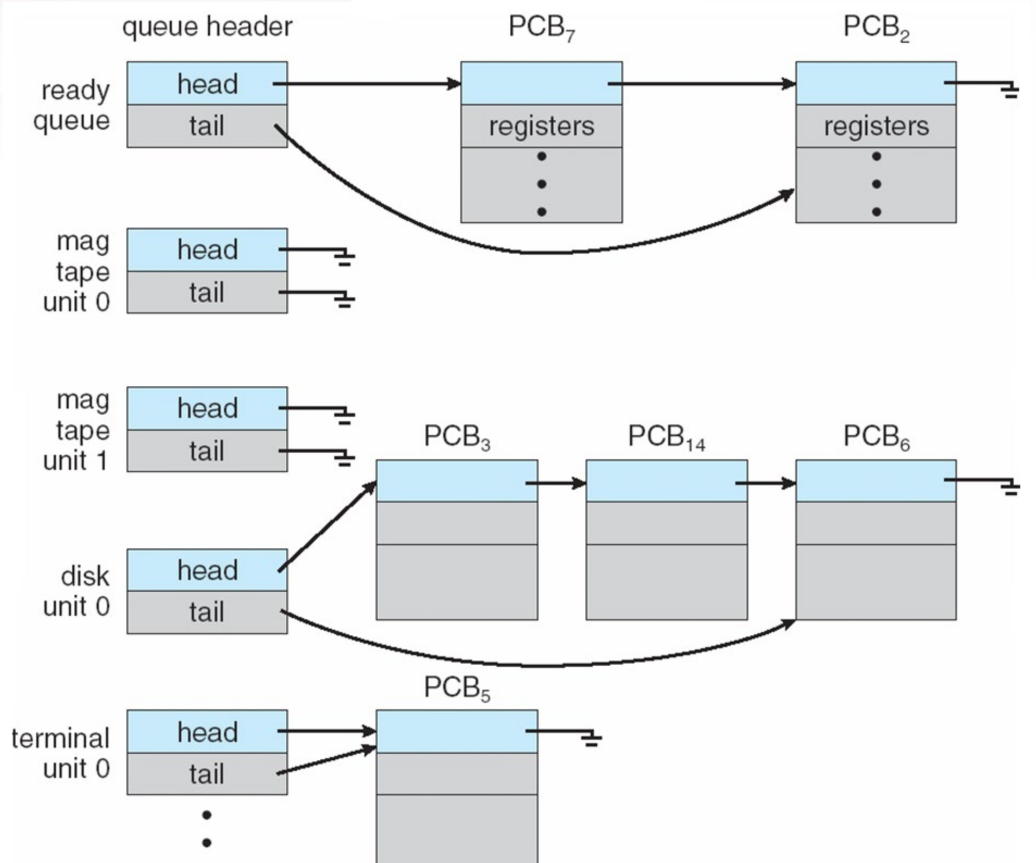


PROCESS CONTROL BLOCK

Queue and various I/O Device Queues



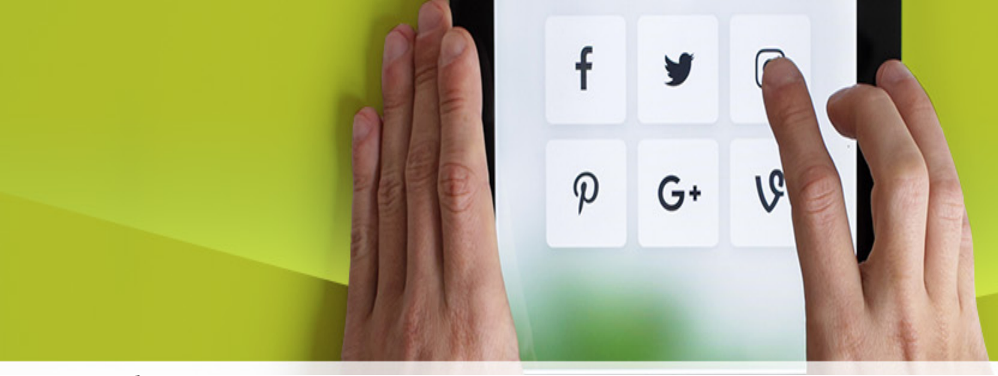
Process state. The state may be new, ready, running, waiting, halted, and so on.
Program counter. The counter indicates the address of the next instruction to be executed for this process
CPU registers. They include accumulators, index registers,stack pointers, and general-purpose registers, plus any condition-code information.
CPU-scheduling information. This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters.
Memory-management information. This information may include such information as the value of the base and registers, the page tables, or the segment tables, depending on the memory system used by the operating system.
Accounting information. This information includes the amount of CPU and real time used, time limits, account numbers, job or process numbers,
I/O status information. This information includes the list of I/O devices allocated to the process, a list of open files, and so on.



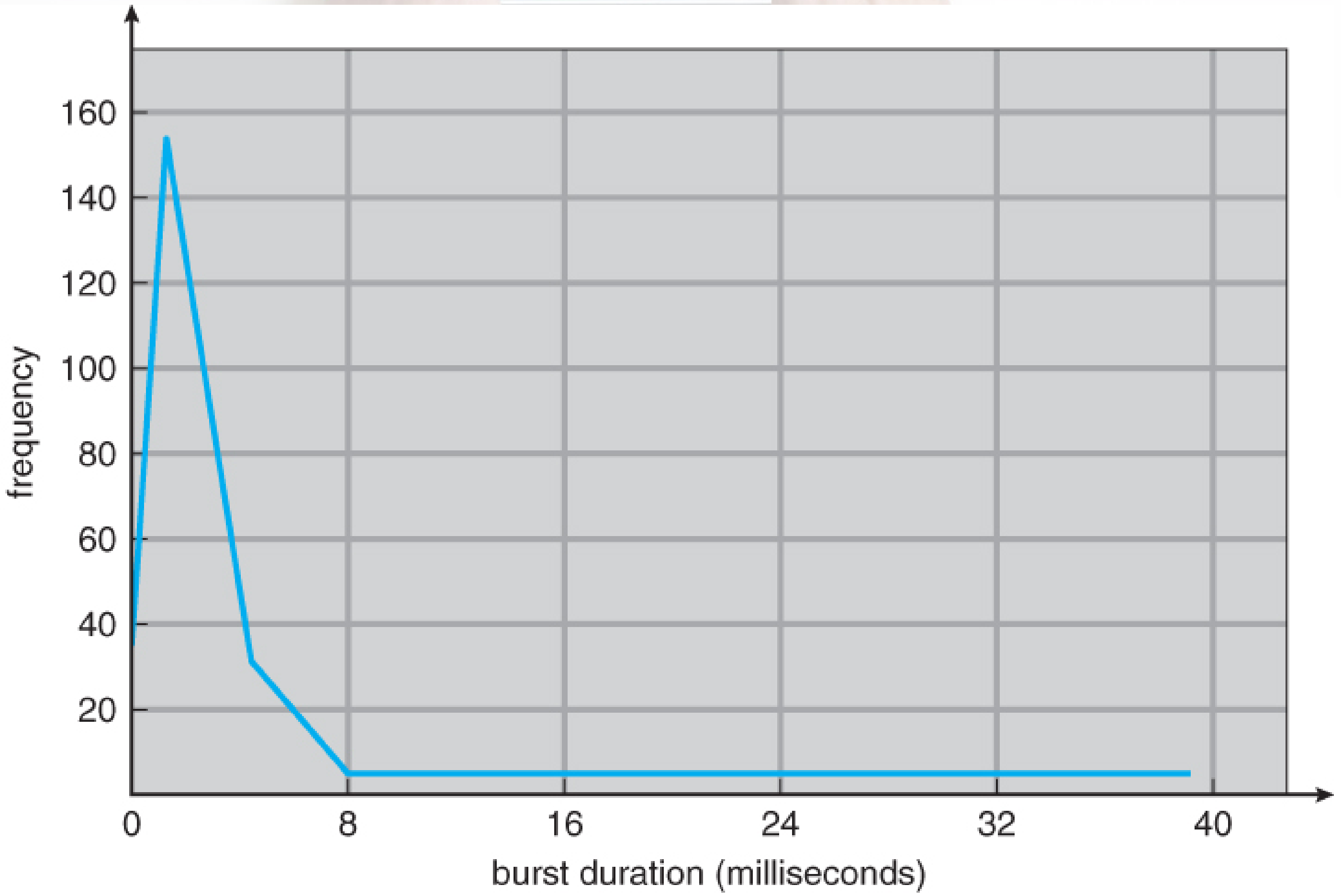
SWITCHING

The CPU-I/O Cycle

- We observe that processes require alternate use of processor and I/O in a repetitive fashion
- Each cycle consist of a CPU burst (typically of 5 ms) followed by a (usually longer) I/O burst
- A process terminates on a CPU burst
- CPU-bound processes have longer CPU bursts than I/O-bound processes

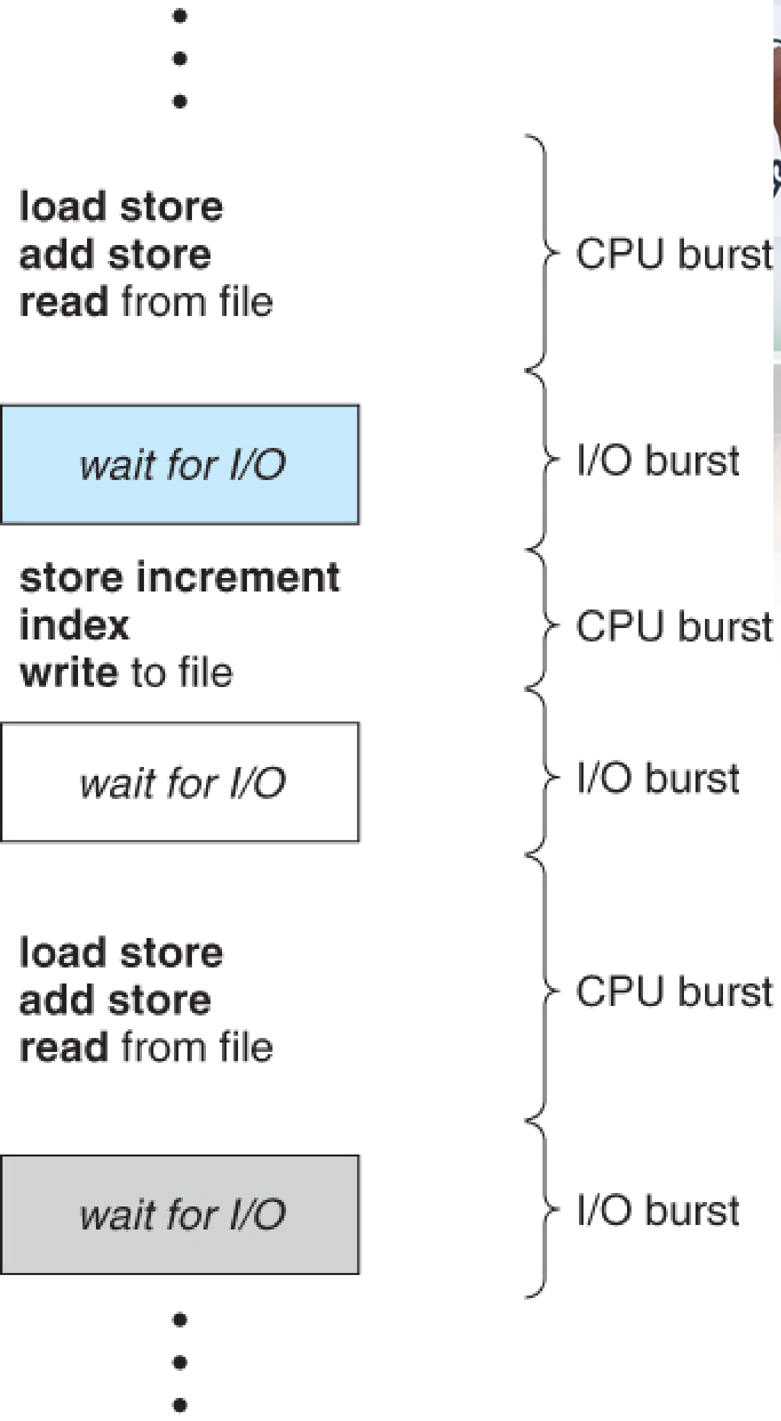


The CPU-I/O Cycle



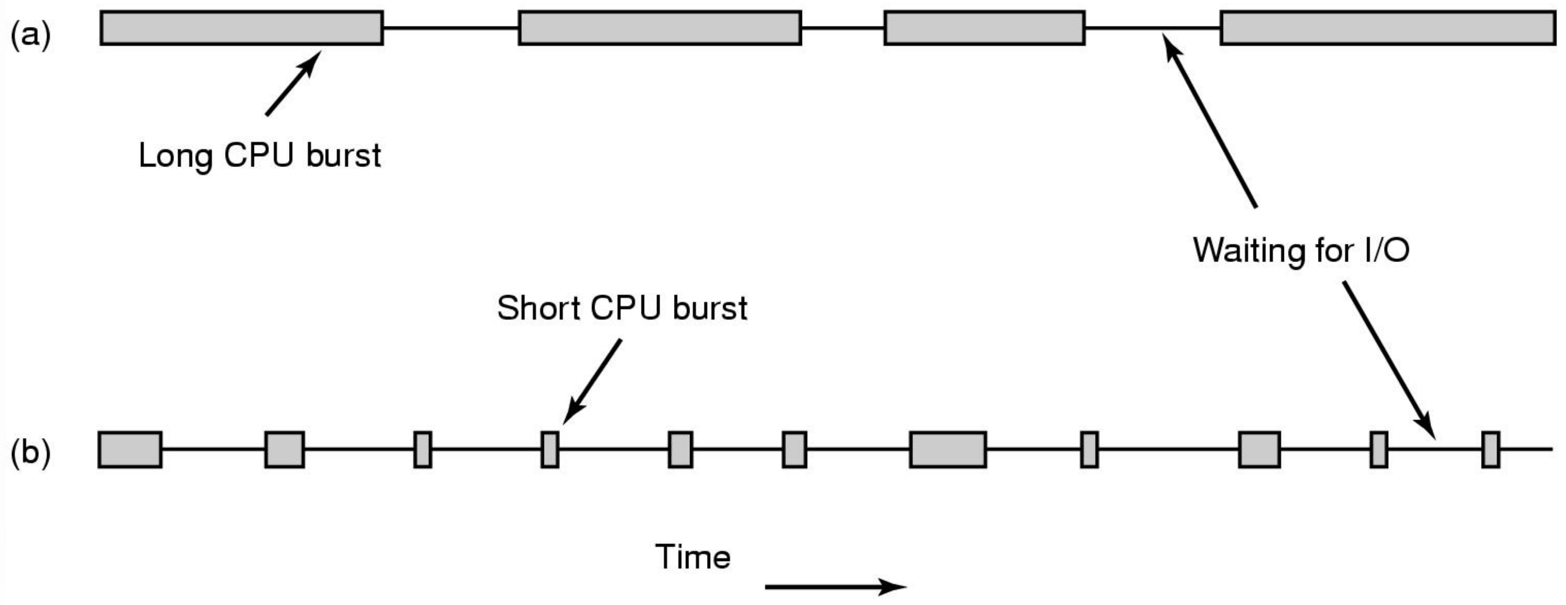
CPU bursts vary from process to process, and from program to program, but an extensive study shows frequency patterns similar to that

The CPU-I/O Cycle



- Almost all processes alternate between two states in a continuing ***cycle :***
  - A CPU burst of performing calculations, and
  - An I/O burst, waiting for data transfer in or out of the system.

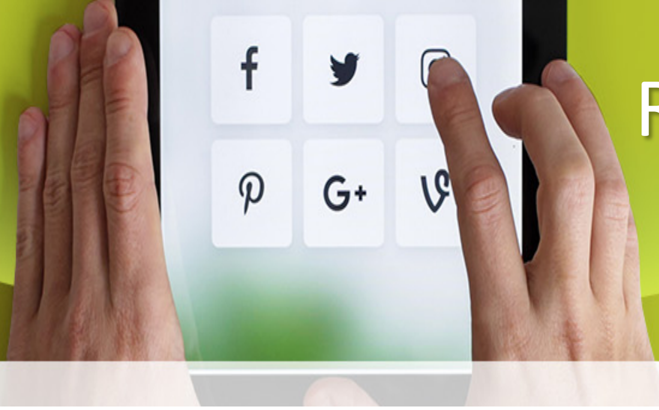
CPU/IO Bursts



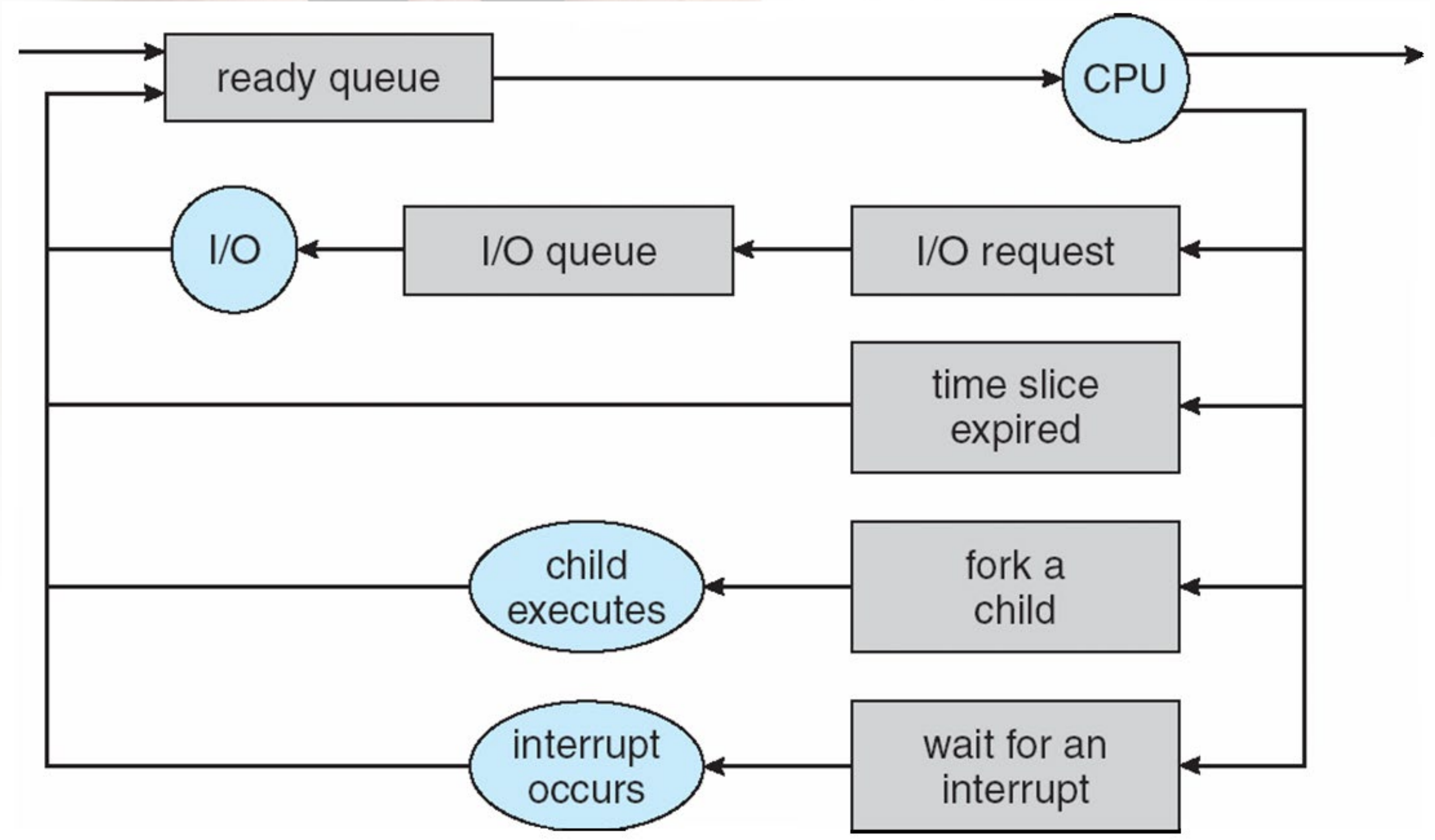
- Bursts of CPU usage alternate with periods of I/O wait
  - a CPU-bound process
  - an I/O bound process

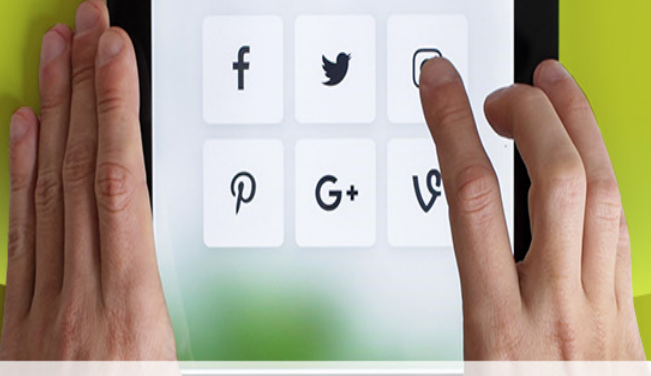
When to Switch a Process?

- A process switch may occur whenever the OS has gained control of CPU. i.e., when:
  - Supervisor Call
    - explicit request by the program (example: file open) – the process will probably be blocked.
  - Trap
    - an error resulted from the last instruction – it may cause the process to be moved to terminated state.
  - Interrupt
    - the cause is external to the execution of the current instruction – control is transferred to Interrupt Handler.



Reasons for Process Switch

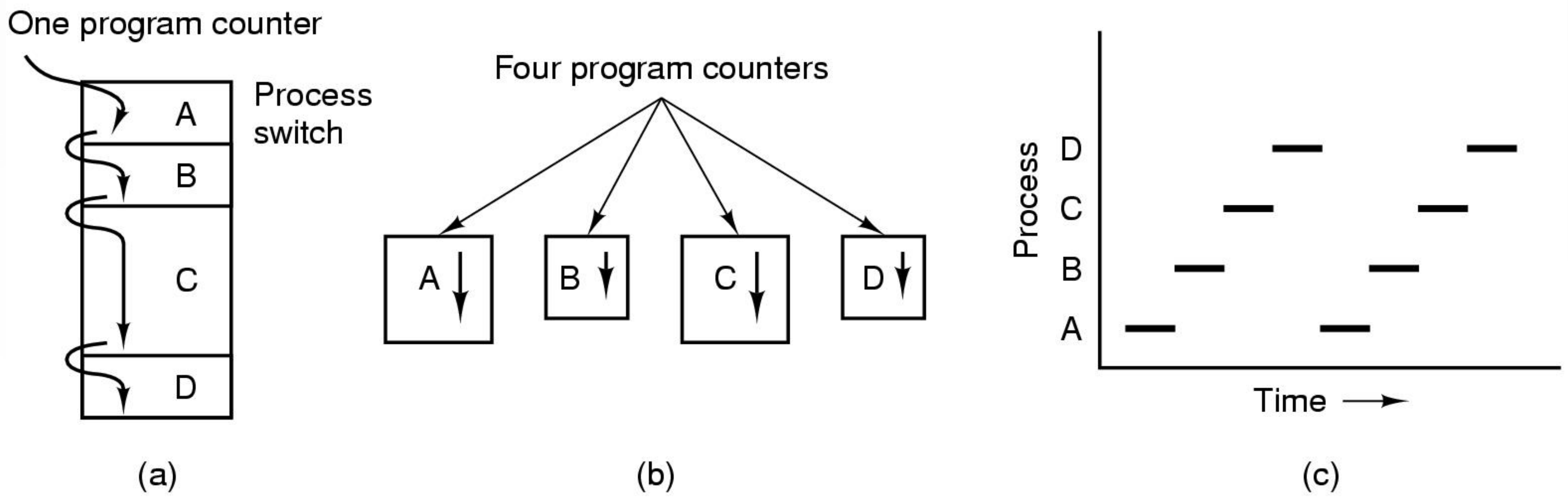


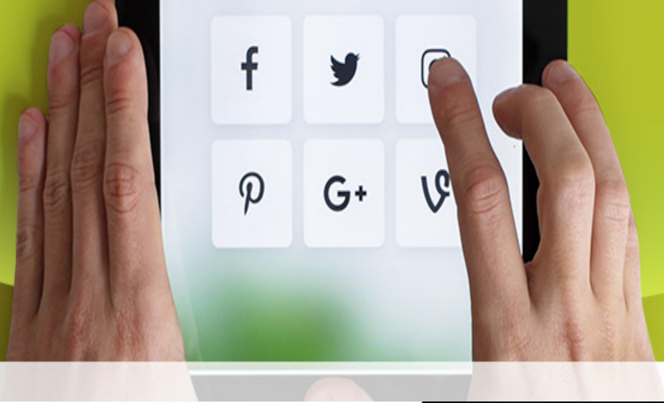


Context Switch

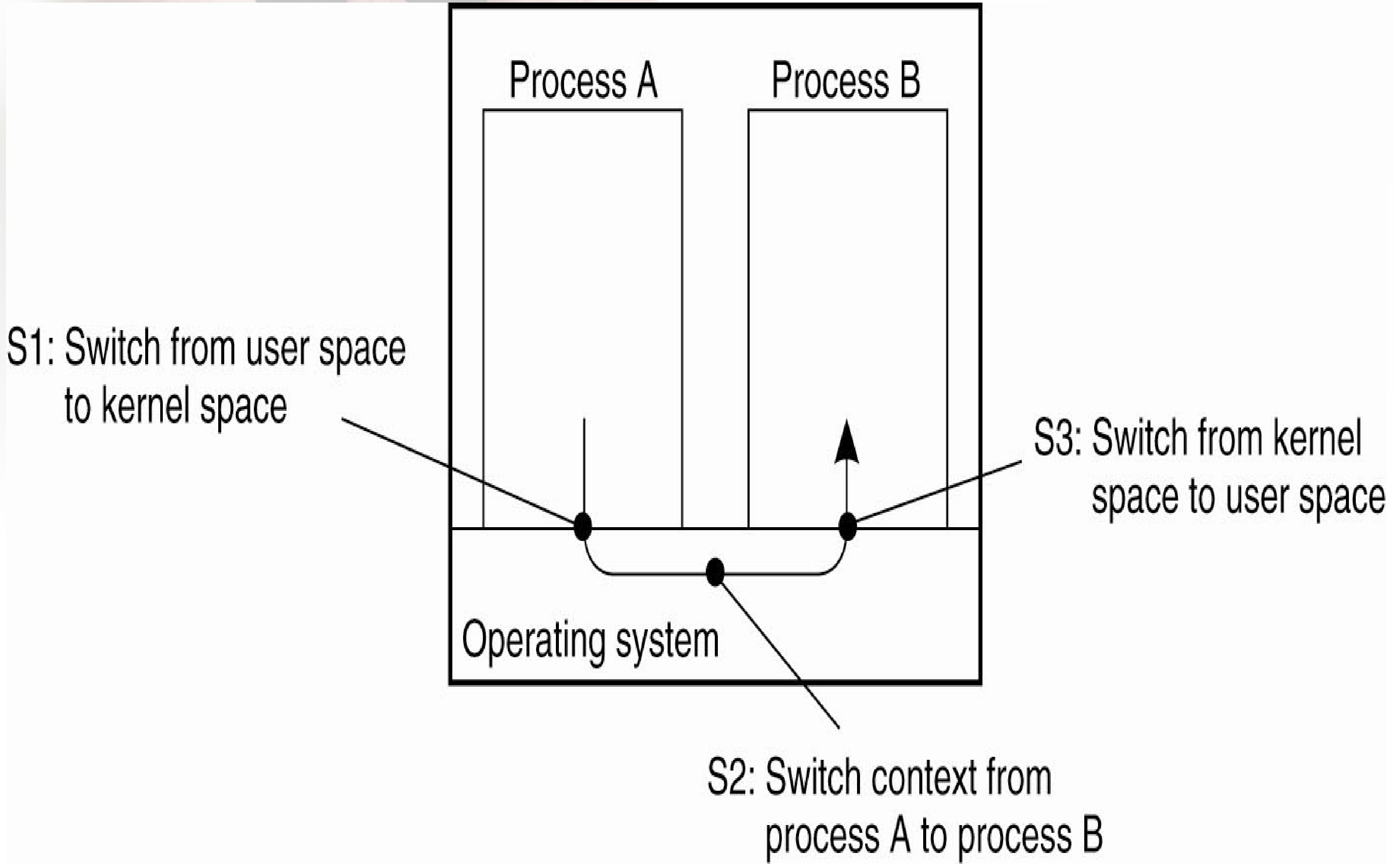
- When CPU switches to another process, the system *must save the state of the old process* and load the saved state for the new process.
- This is called context switch.
- Context of a process represented in the PCB.
- The time it takes is dependent on hardware support.
- Context-switch time is overhead; the system does no useful work while switching.

Process Switch

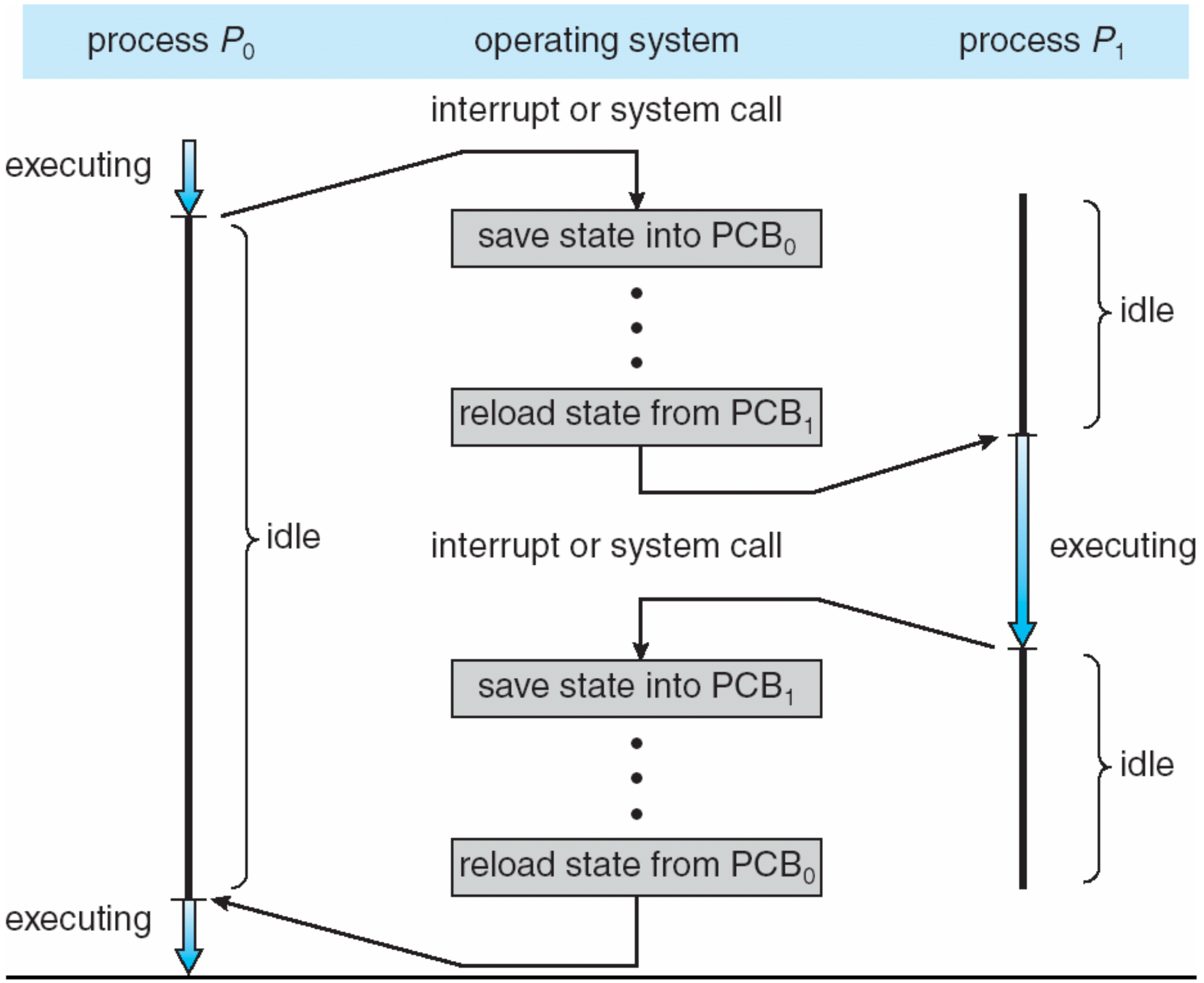




Context switch between processes (1)


Process A
Process B
S1: Switch from user space to kernel space
S3: Switch from kernel space to user space
Operating system
S2: Switch context from process A to process B

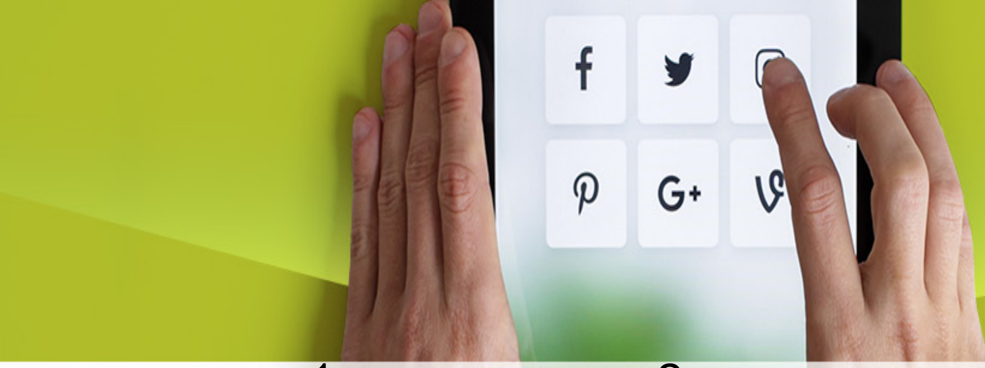
Context switch between processes (2)

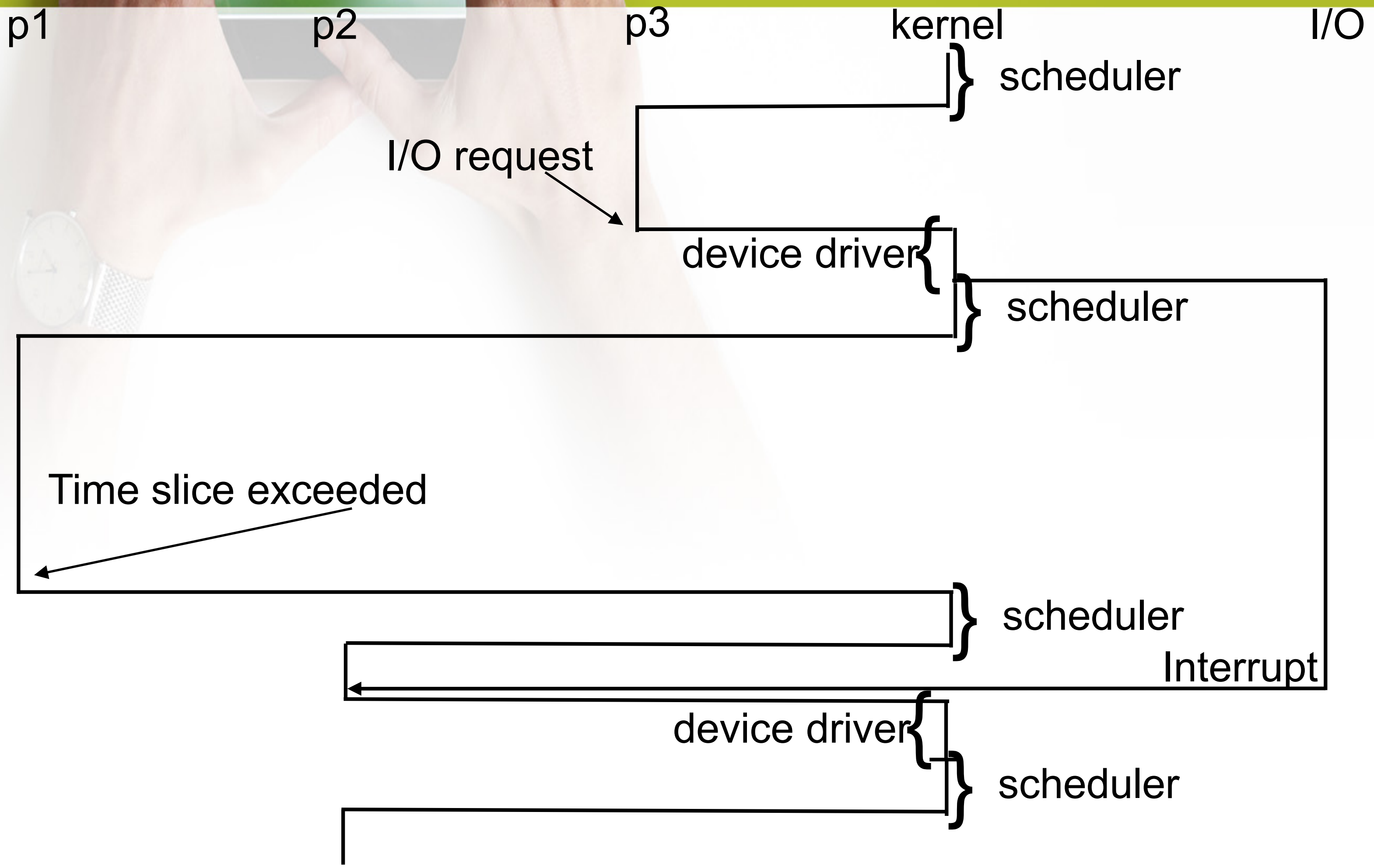


Steps in Context Switch

- Save context of processor including program counter and other registers.
- Update the PCB of the running process with its new state and other associate information.
- Move PCB to appropriate queue – ready, blocked,
- Select another process for execution.
- Update PCB of the selected process.
- Restore CPU context from that of the selected process.

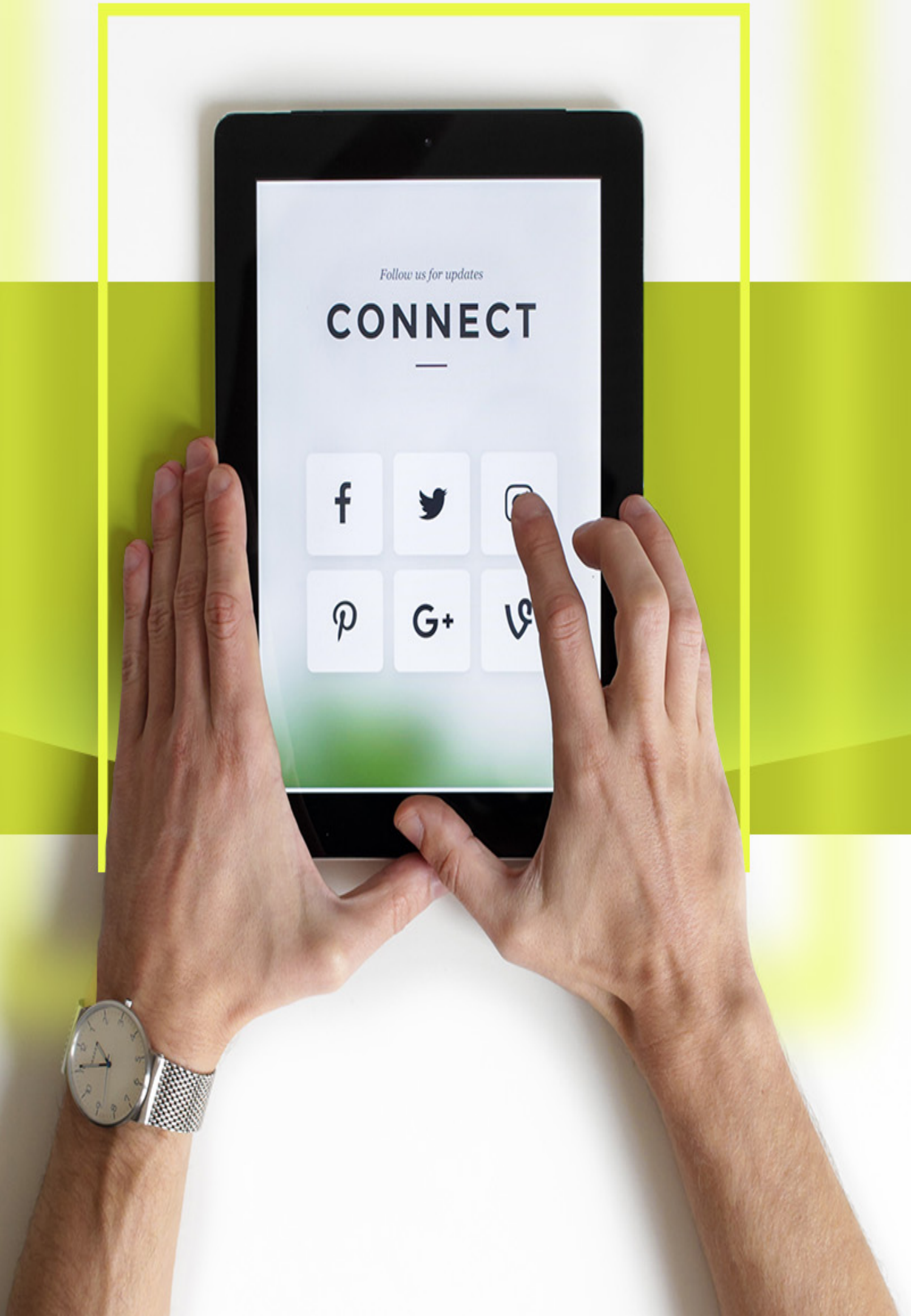
Example of Context Switch



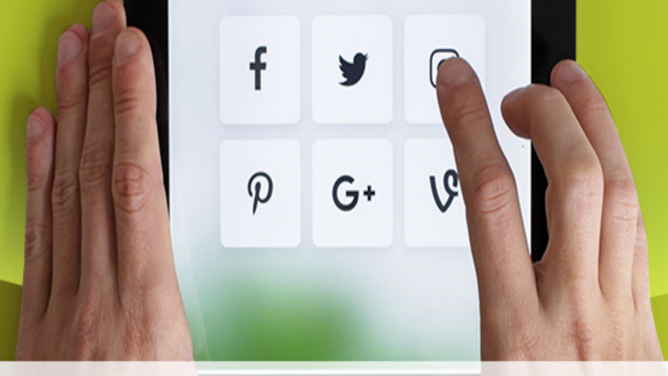


Mode Switch

- It may happen that an interrupt does not produce a context switch.
- The control can just return to the interrupted program.
- Then only the processor state information needs to be saved on stack.
- This is called mode switch (user to kernel mode when going into Interrupt Handler).
- Less overhead: <u>no need to update the PCB </u>like for context switch.



Scheduling Algorithms



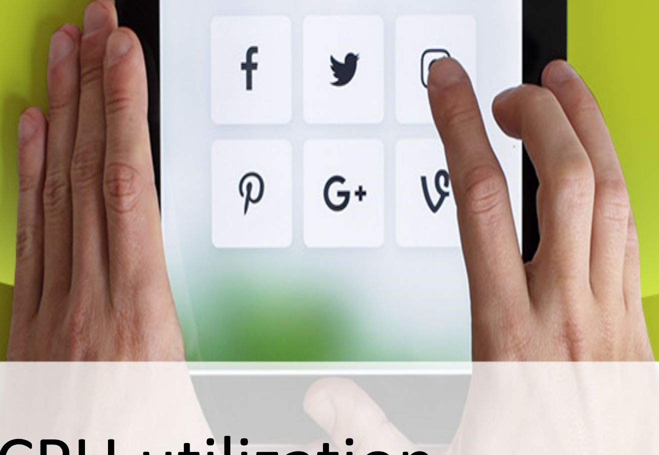
Topics for discussion

- Various algorithms
  - First-come, first-served
  - Priority queues
  - Round-robin

Scheduling in Linux

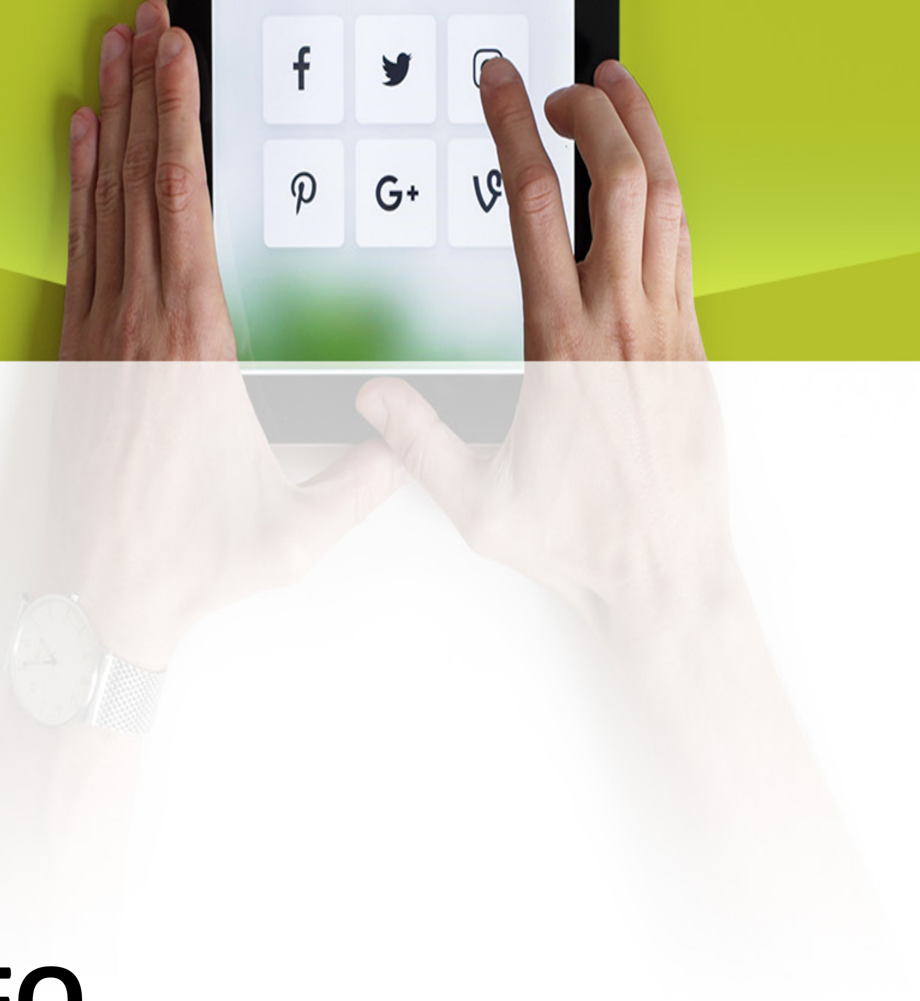
Three classes of threads for scheduling purposes:

- Real-time FIFO
- Real-time round robin
- Timesharing (for all non real-time processes)



Optimization Criteria

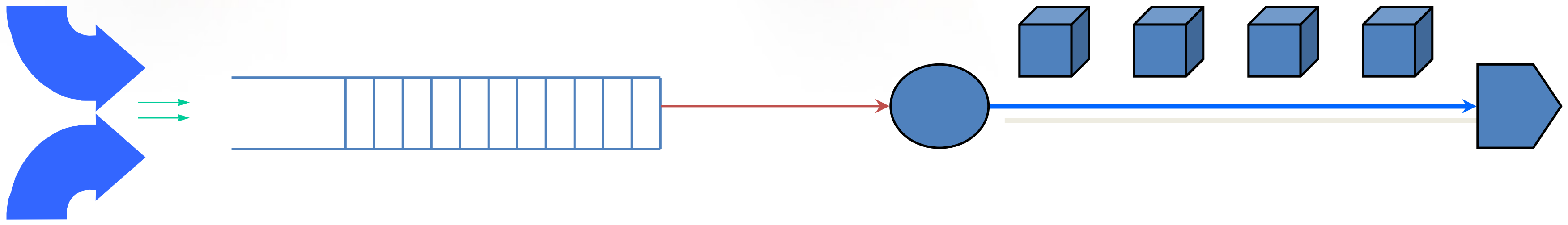
- Max CPU utilization
- Max throughput
- Min turnaround time
- Min waiting time
- Min response time

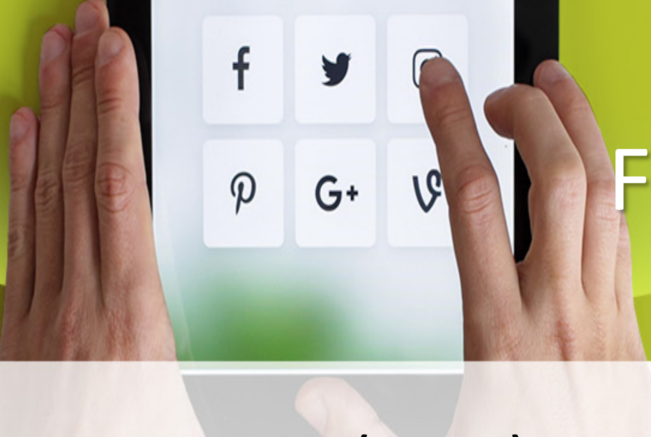


FIFO

FIFO Queuing

- Simplest Algorithm, widely used.
- Scheduling is done using first-in first-out (FIFO) discipline
- All flows are fed into the same queue





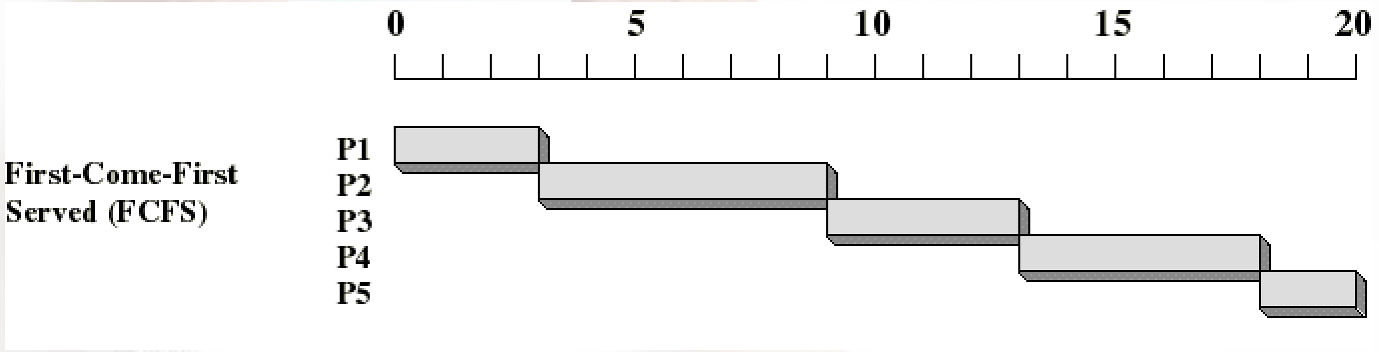
FIFO Queuing (cont'd)

- First-In First-Out (FIFO) queuing
  - First Arrival, First Transmission
  - Completely dependent on arrival time
  - No notion of priority or allocated buffers
  - No space in queue, packet discarded
  - Flows can interfere with each other; No isolation; malicious monopolization;

FCFS drawbacks

- Favors CPU-bound processes
  - A CPU-bound process monopolizes the processor
  - I/O-bound processes have to wait until completion of CPU-bound process
    - I/O-bound processes may have to wait even after their I/Os are completed (poor device utilization)
  - Better I/O device utilization could be achieved if I/O bound processes had higher priority

First Come First Served (FCFS)

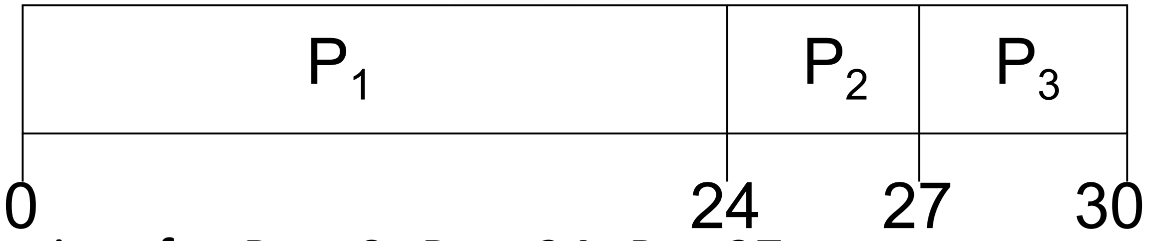


- Selection function: the process that has been waiting the longest in the ready queue (hence, FCFS)
- Decision mode: non-preemptive
  - a process runs until it blocks for an I/O

First-Come, First-Served (FCFS) Scheduling

Process	Burst Time
P_1	24
P_2	3
P_3	3

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



Waiting time for P_1 = 0; P_2 = 24; P_3 = 27

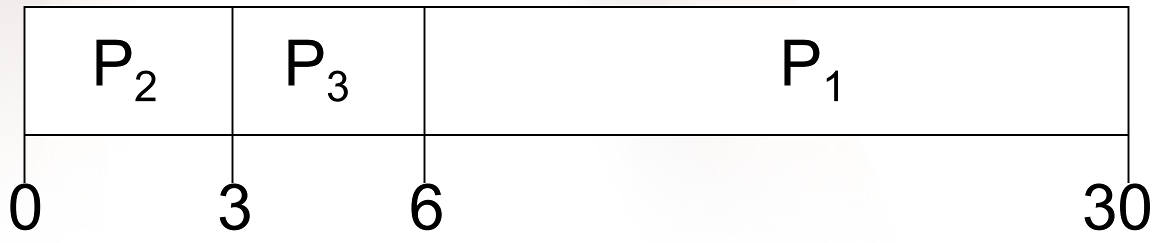
- Average waiting time: (0 + 24 + 27)/3 = 17

FCFS Scheduling (Cont.)

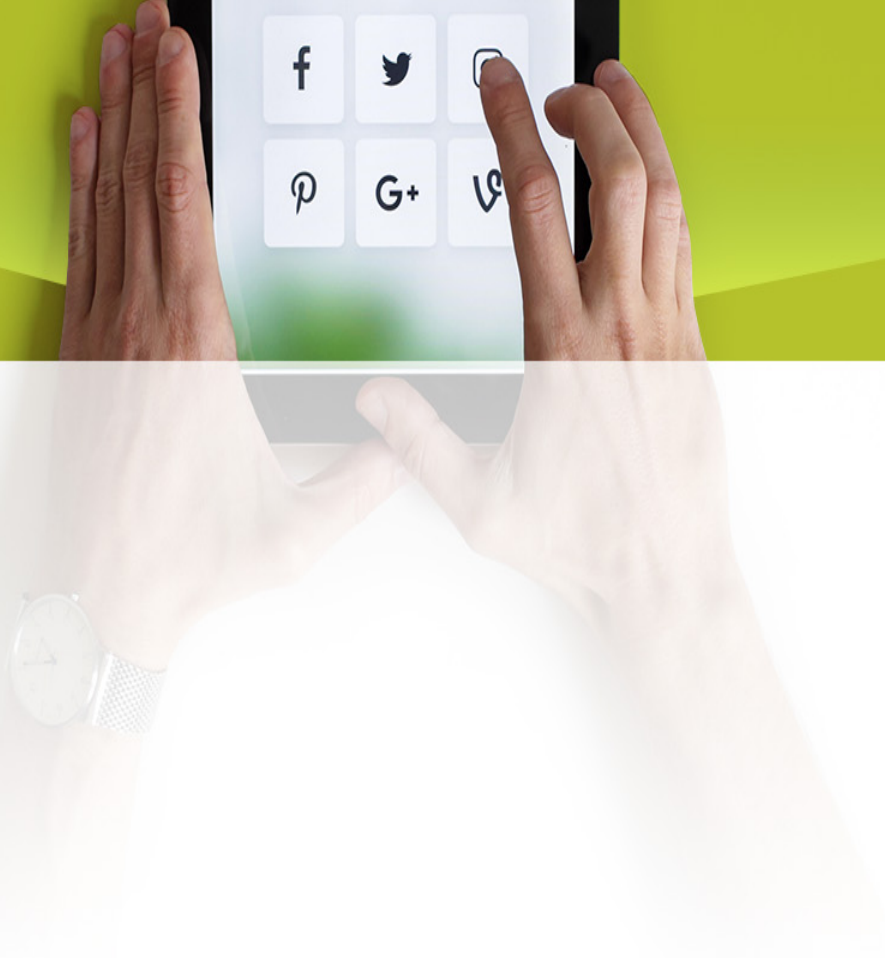
Suppose that the processes arrive in the order

P_2 , P_3 , P_1 .

- The Gantt chart for the schedule is:

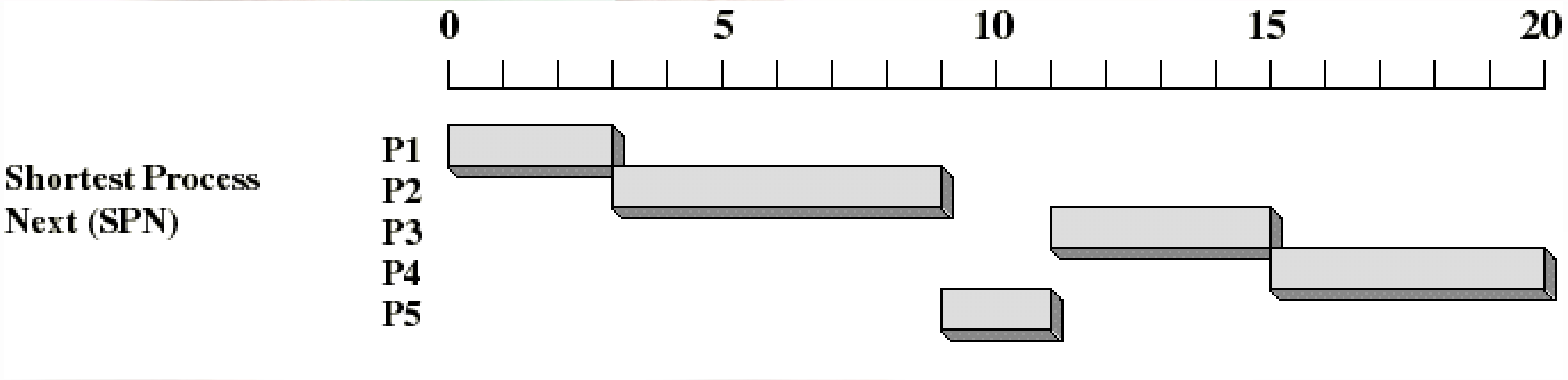


- Waiting time for $P_1 = 6$; $P_2 = 0$; $P_3 = 3$
- Average waiting time: (6 + 0 + 3)/3 = 3
- Much better than previous case.
- *Convoy effect* short process behind long process

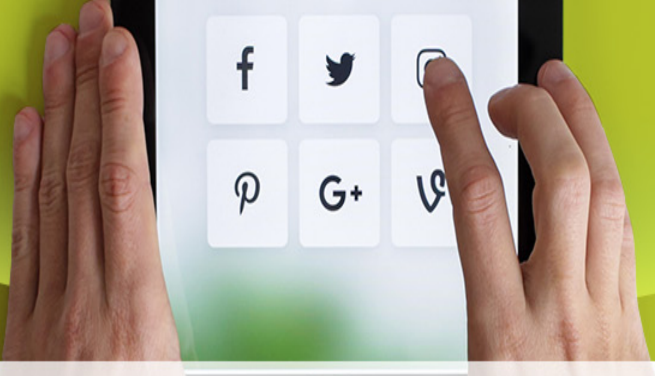


SHORTEST JOB FIRST

Shortest Job First (Shortest Process Next)

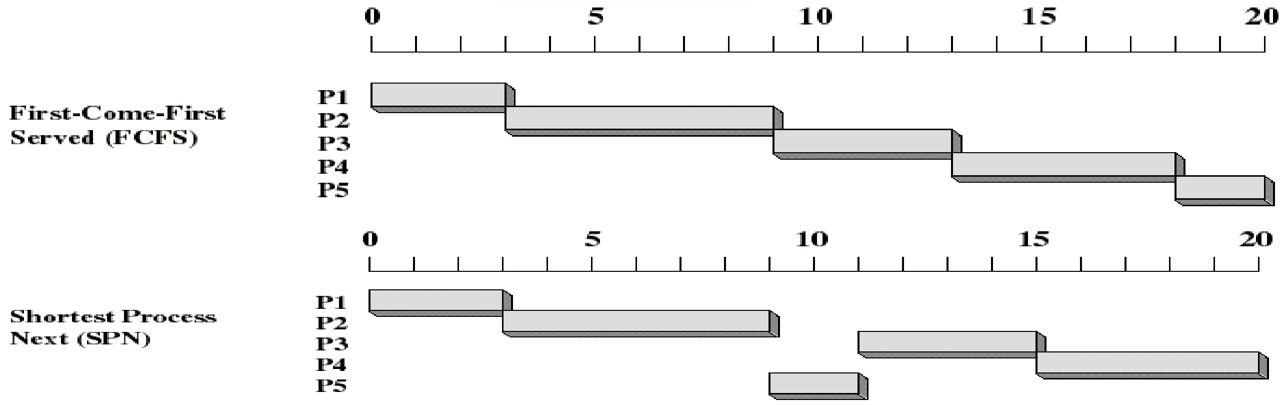


- Selection function: the process with the shortest expected CPU burst time
  - I/O-bound processes will be selected first
- Decision mode: non-preemptive
- The required processing time, i.e., the CPU burst time, must be estimated for each process



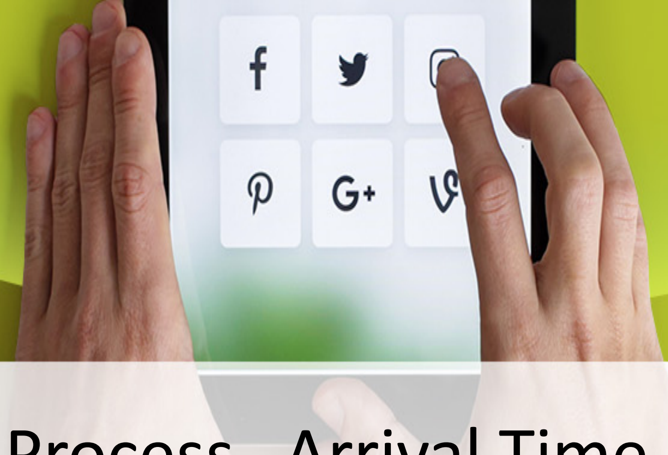
Is SJF/SPN optimal?

- If the metric is turnaround time (response time), is SJF or FCFS better?
- For FCFS, resp_time=(3+9+13+18+20)/5 = ?
  - Note that Rfcfs = 3+(3+6)+(3+6+4)+…. = ?
- For SJF, resp_time=(3+9+11+15+20)/5 = ?
  - Note that Rfcfs = 3+(3+6)+(3+6+4)+…. = ?
- Which one is smaller? Is this always the case?



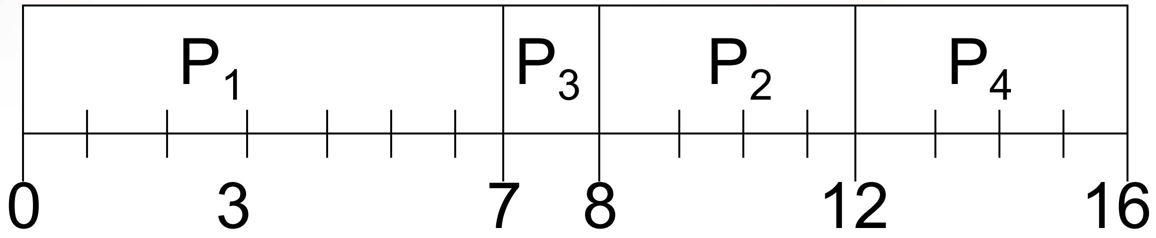
Is SJF/SPN optimal?

- Take each scheduling discipline, they both choose the same subset of jobs (first *k* jobs).
- At some point, each discipline chooses a different job (FCFS chooses *k1* SJF chooses *k2*)
- $R_{fcfs}=nR_1+(n-1)R_2+\ldots+(n-k1)R_{k1}+\ldots.+(n-k2)\ R_{k2}+\ldots.+R_n$
- $R_{sjf}=nR_1+(n-1)R_2+\ldots+(n-k2)R_{k2}+\ldots.+(n-k1)\ R_{k1}+\ldots.+R_n$
- Which one is smaller? R_{fcfs} or R_{sjf}?



Example of Non-Preemptive SJF

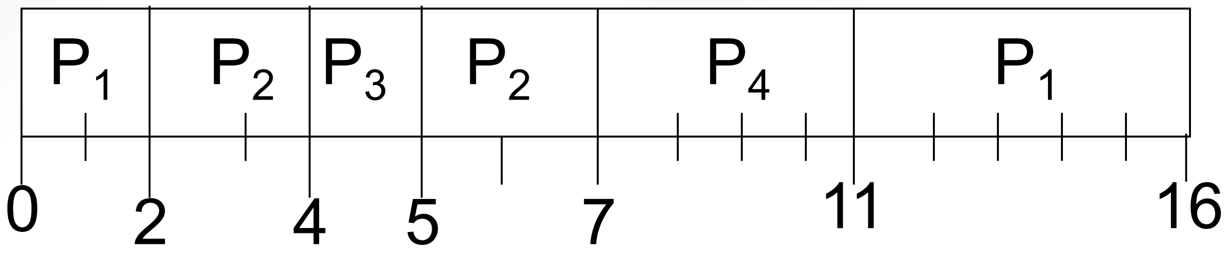
Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4



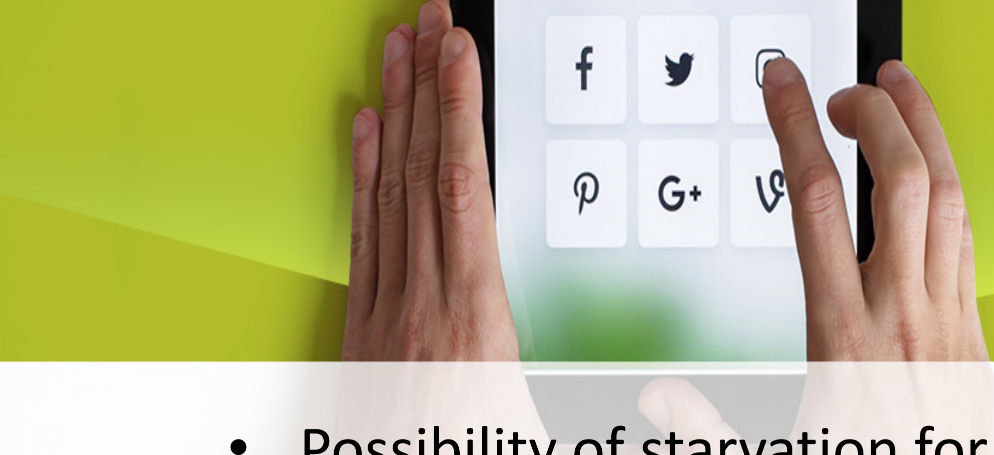
- Average waiting time = (0 + 6 + 3 + 7)/4 = 4

Example of Preemptive SJF

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

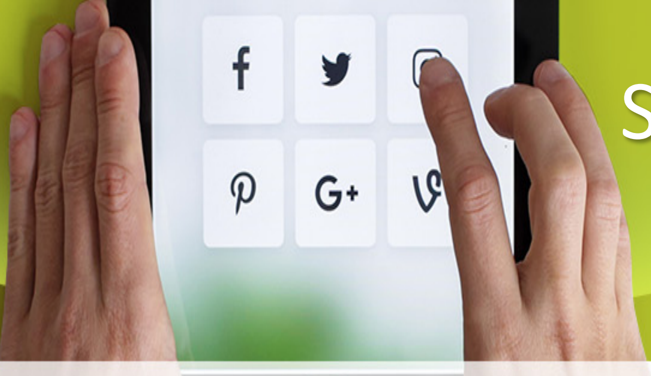


- Average waiting time = (9 + 1 + 0 +2)/4 = 3



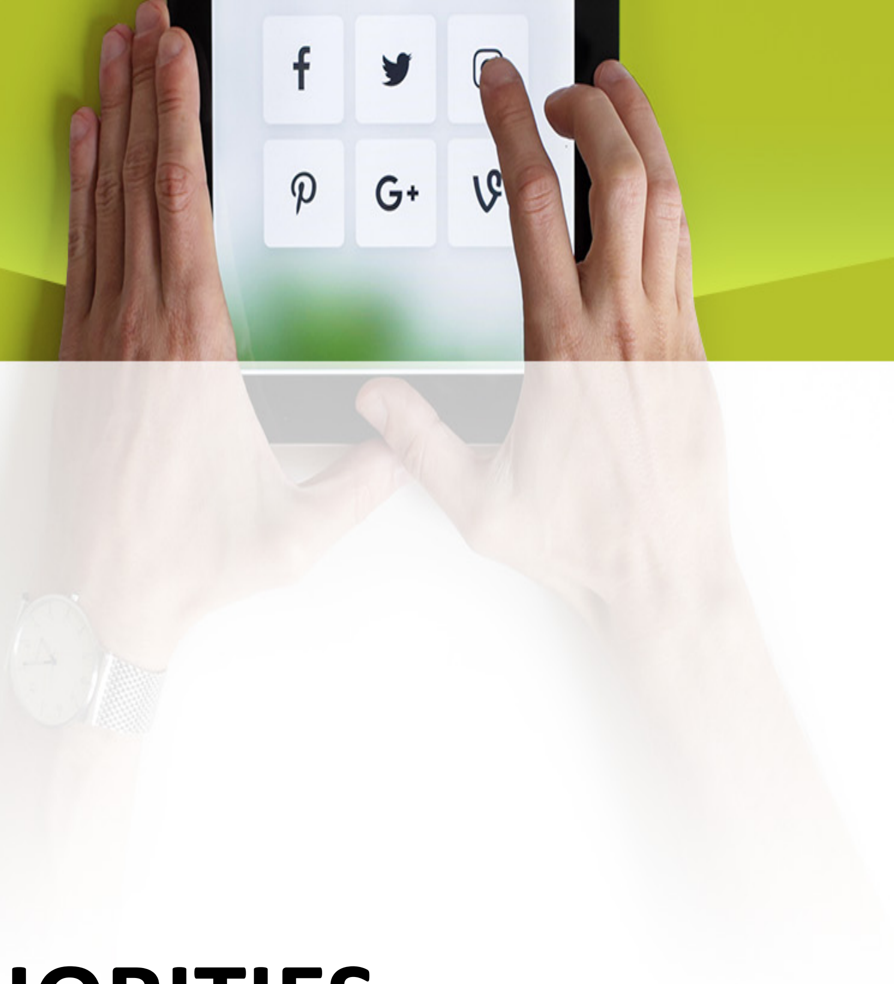
SJF / SPN Critique

- Possibility of starvation for longer processes
- Lack of preemption is not suitable in a time sharing environment
- SJF/SPN implicitly incorporates priorities
  - Shortest jobs are given preferences
  - CPU bound process have lower priority, but a process doing no I/O could still monopolize the CPU if it is the first to enter the system



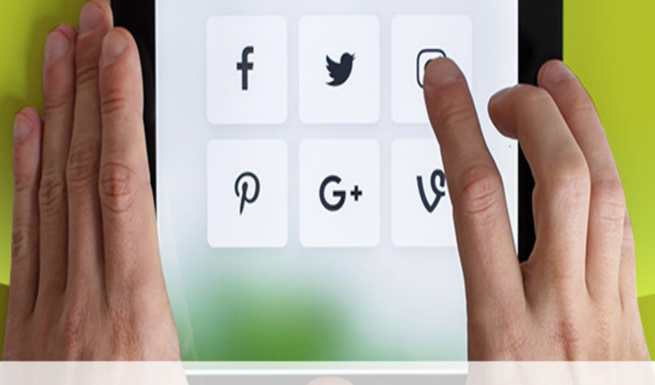
Shortest job first: critique

- Possibility of starvation for longer processes as long as there is a steady supply of shorter processes
- Lack of preemption is not suited in a time sharing environment
  - CPU bound process gets lower priority (as it should) but a process doing no I/O could still monopolize the CPU if he is the first one to enter the system
- SJF implicitly incorporates priorities: shortest jobs are given preferences
- The next (preemptive) algorithm penalizes directly longer jobs



PRIORITIES

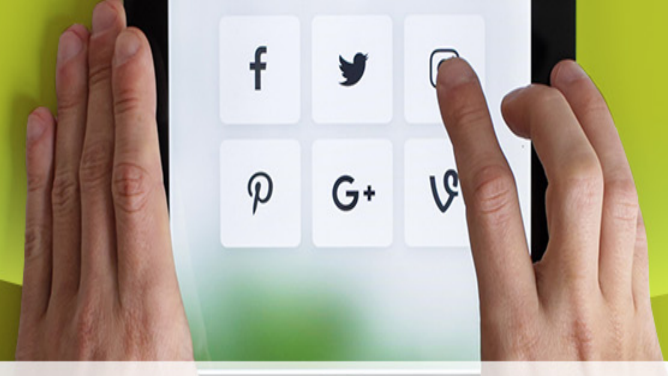
Priority Scheduling



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

Priorities

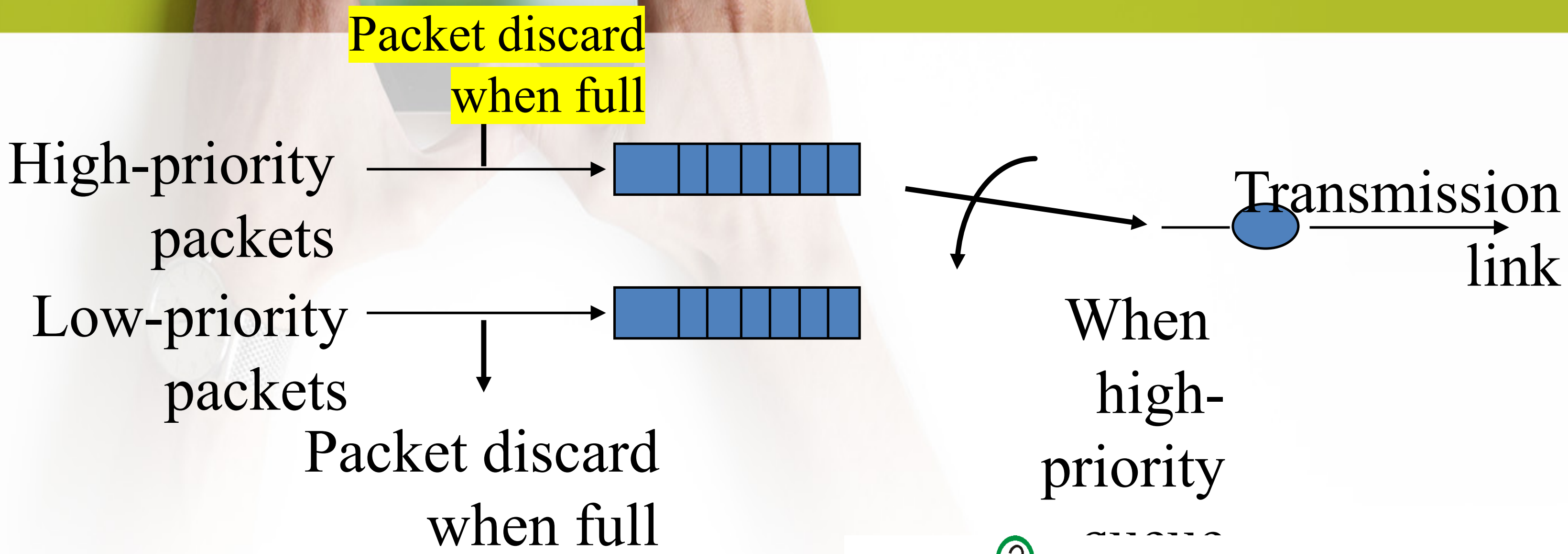
- Implemented by having multiple ready queues to represent each level of priority
- Scheduler the process of a higher priority over one of lower priority
- Lower-priority may suffer starvation
- To alleviate starvation allow dynamic priorities
  - The priority of a process changes based on its age or execution history

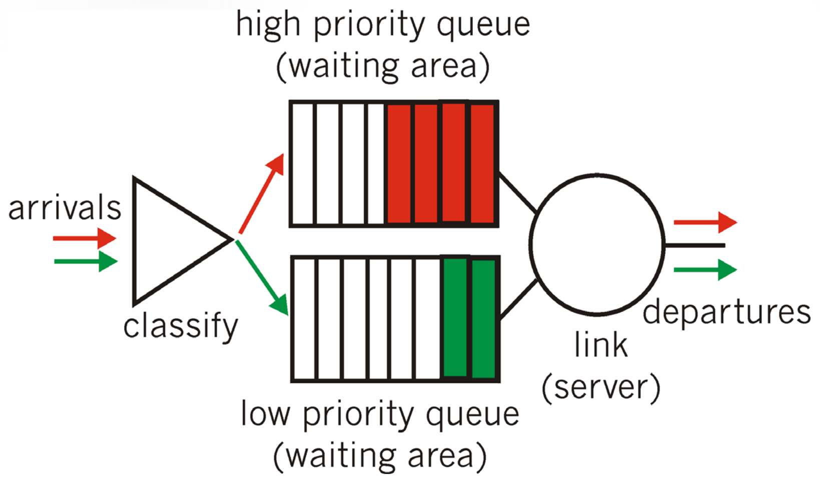


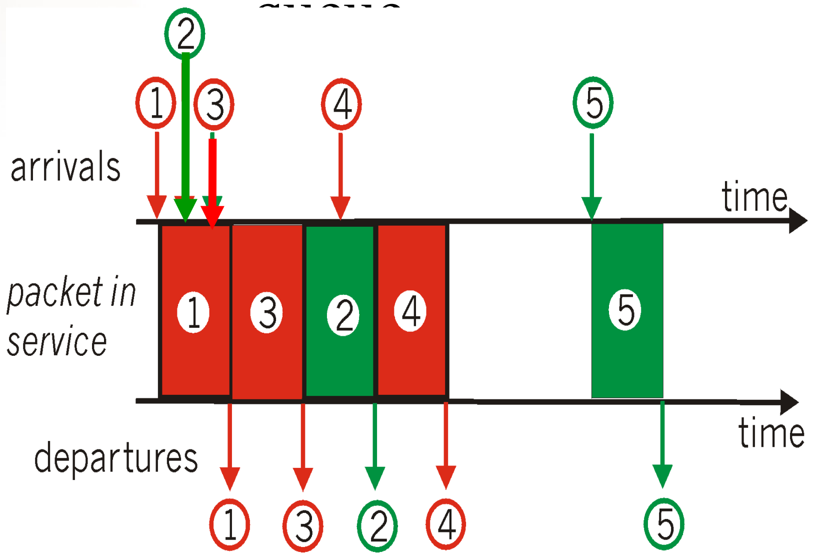
Priority Queuing

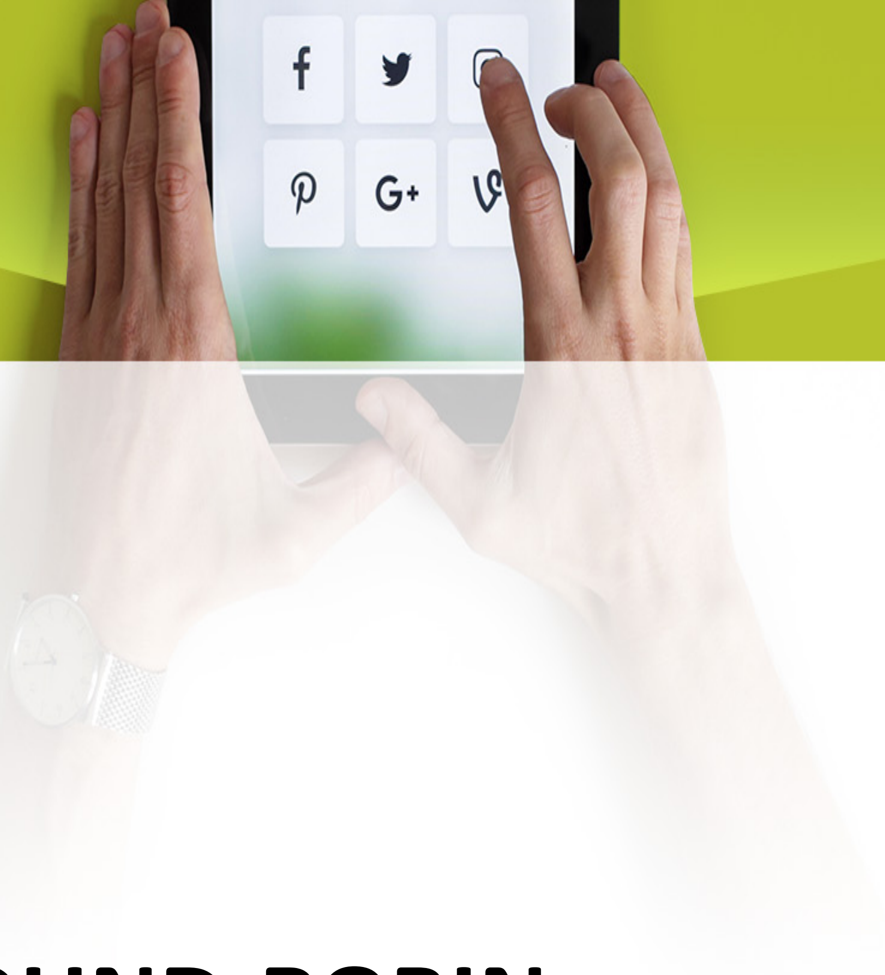
- A priority index is assigned to each packet upon arrival
- Packets transmitted in ascending order of priority index.
  - Priority 0 through n-1
  - Priority 0 is always serviced first
- Priority *i* is serviced only if *0* through *i-1* are empty
- Highest priority has the
  - lowest delay,
  - highest throughput,
  - lowest loss
- Lower priority classes may be starved by higher priority
- Preemptive and non-preemptive versions.

Priority Queuing

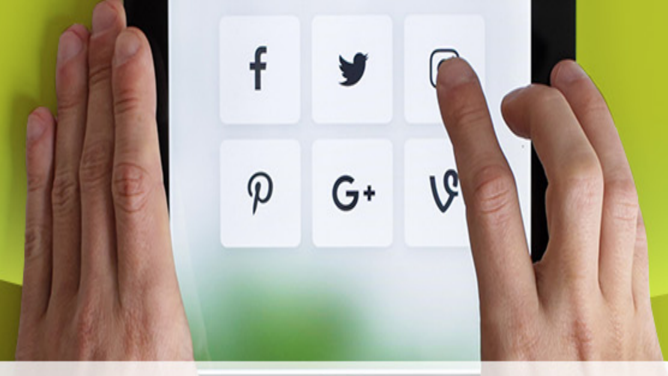



high priority queue (waiting area)
arrivals
classify
link (server)
departures
low priority queue (waiting area)



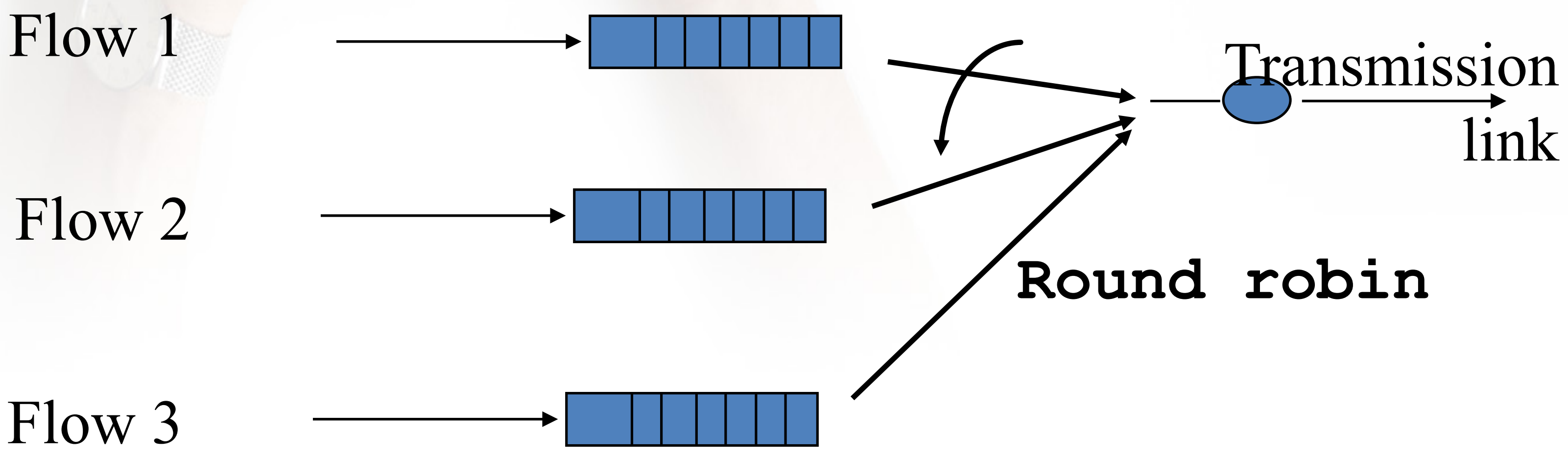


ROUND-ROBIN

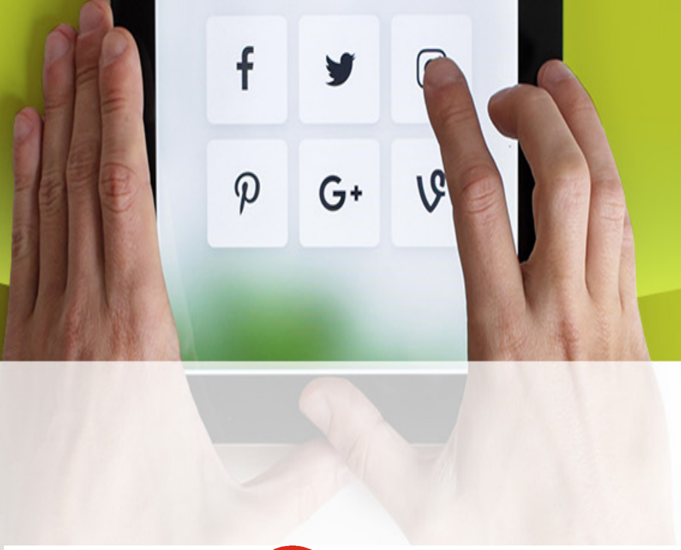


Round Robin: Architecture

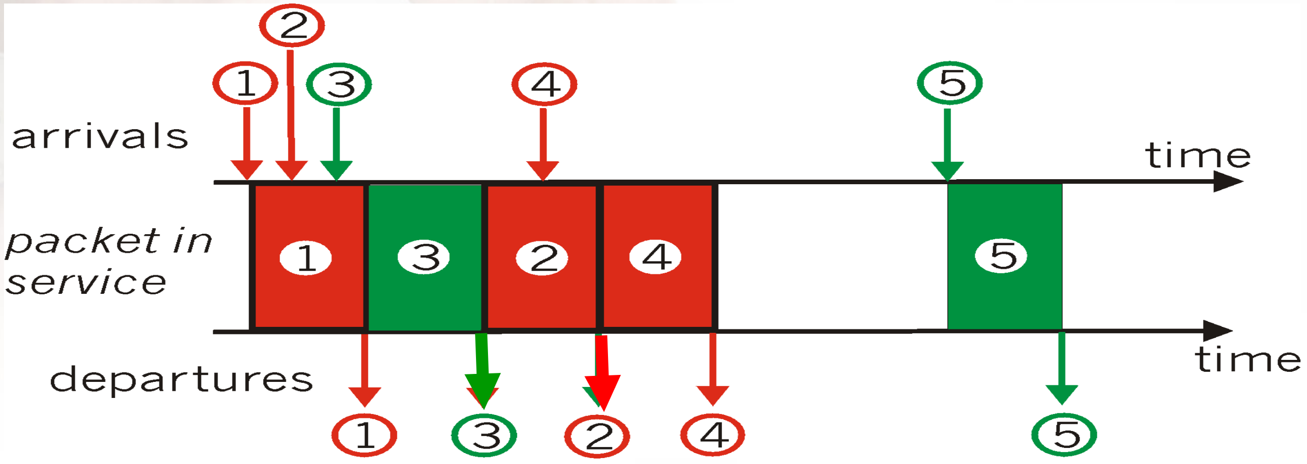
Round Robin: scan class queues serving one from each class that has a non-empty queue



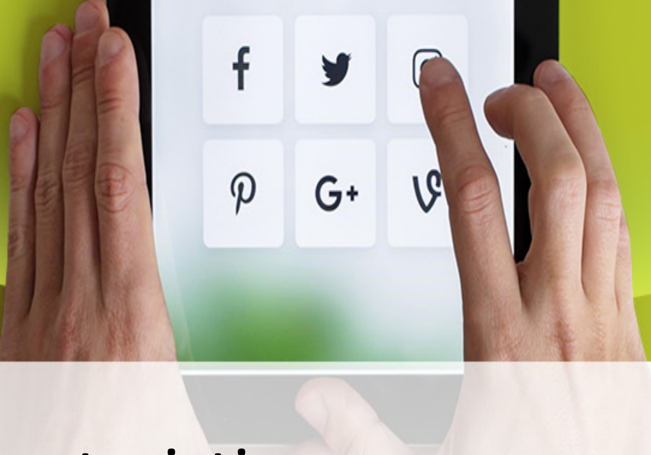
Hardware requirement:
Jump to next non-empty queue



Round Robin Scheduling



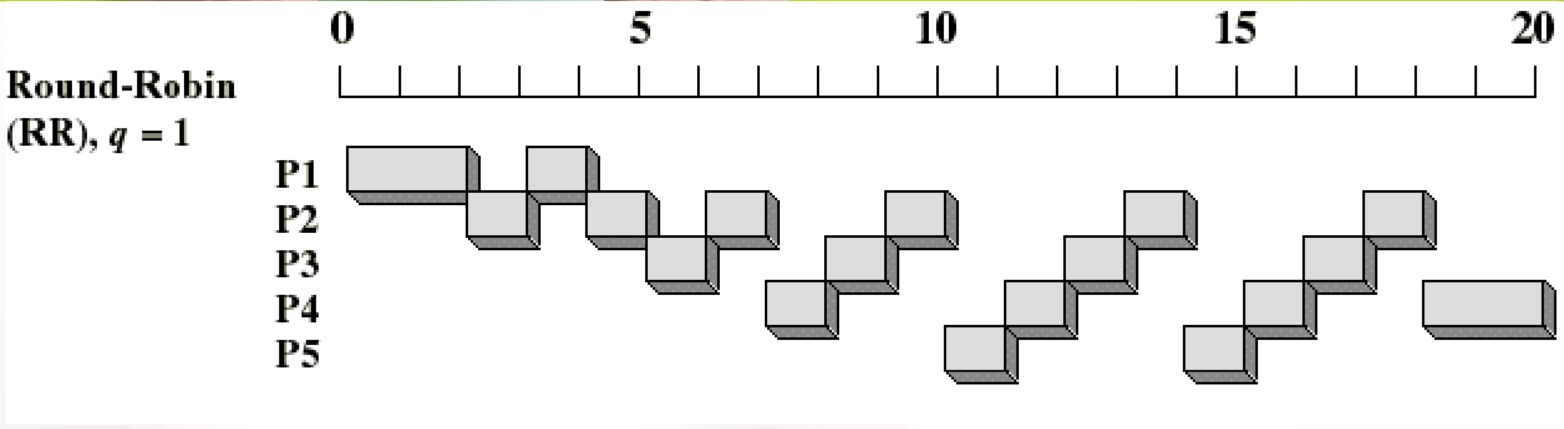
- Round Robin: scan class queues serving one from each class that has a non-empty queue



Round Robin (cont’d)

- Characteristics:
  - Classify incoming traffic into flows (source-destination pairs)
  - Round-robin among flows
- Problems:
  - Ignores packet length (GPS, Fair queuing)
  - Inflexible allocation of weights (WRR,WFQ)
- Benefits:
  - protection against heavy users (why?)

Round-Robin



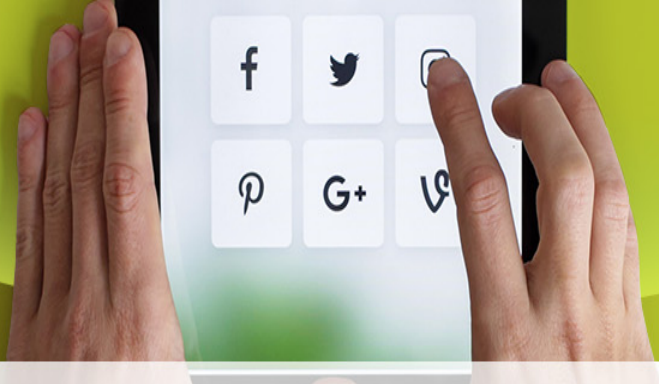
- Selection function: same as FCFS
- Decision mode: preemptive
  - a process is allowed to run until the time slice period (quantum, typically from 10 to 100 ms) has expired
  - a clock interrupt occurs and the running process is put on the ready queue

R-R Time Quantum

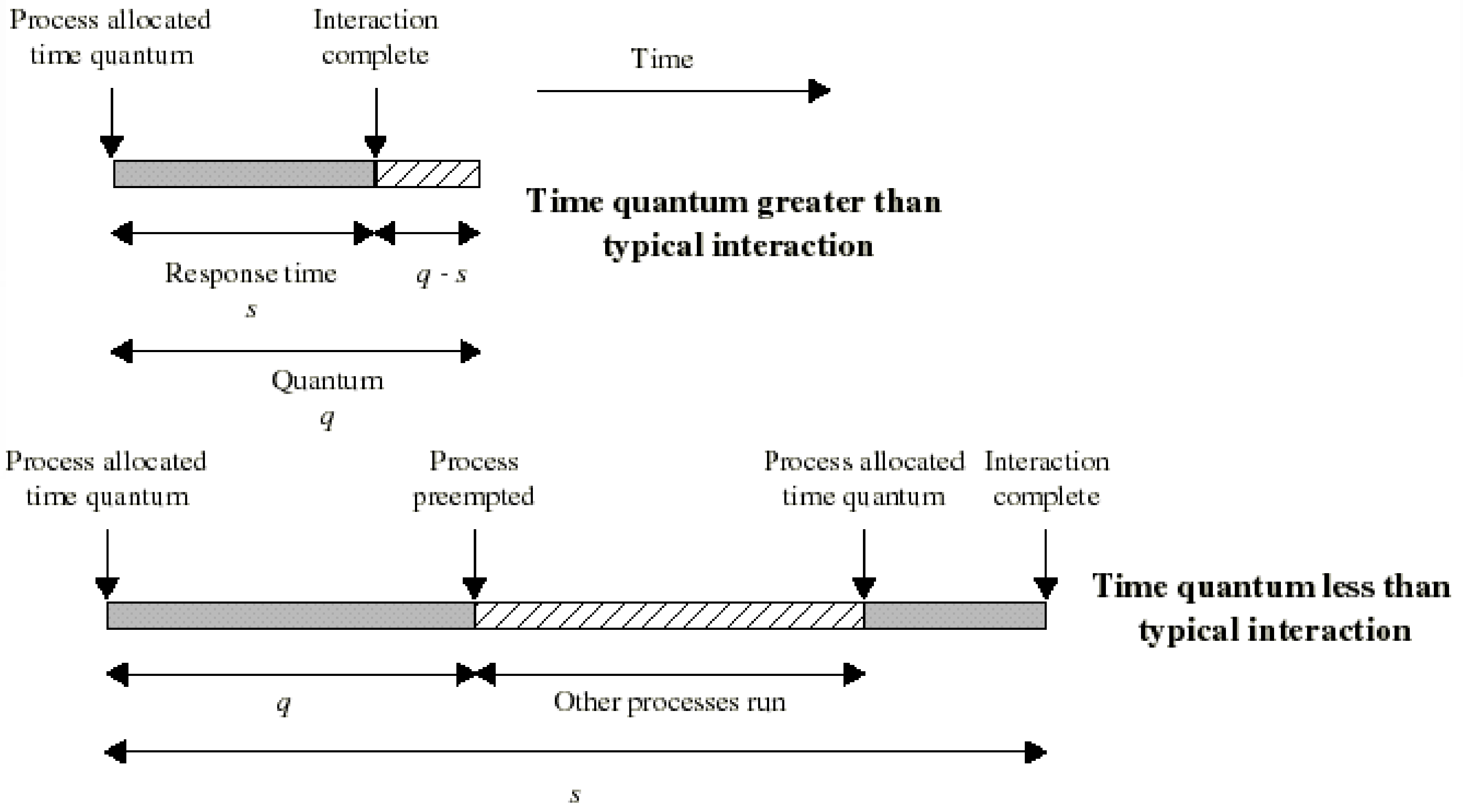
- Quantum must be substantially larger than the time required to handle the clock interrupt and dispatching
- Quantum should be larger then the typical interaction
  - but not much larger, to avoid penalizing I/O bound processes

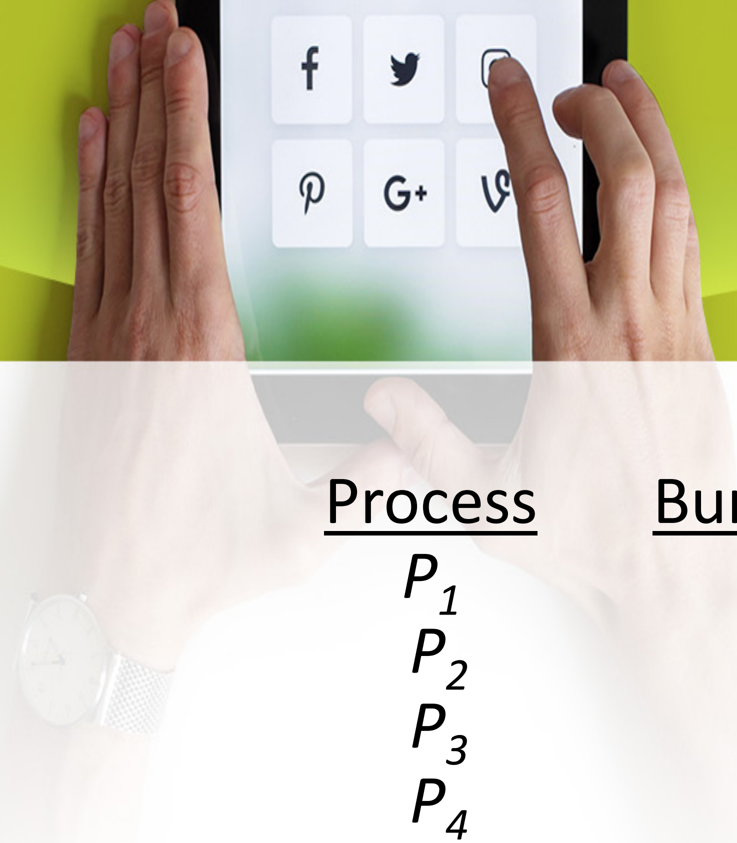
Round Robin (RR)

- Each process gets a small unit of CPU time (*time quantum*), usually 10-100 milliseconds.
  After this time has elapsed, the process is preempted and added to the end of the ready queue.
- If there are *n* processes in the ready queue and the time quantum is *q*, then each process gets 1/*n* of the CPU time in chunks of at most *q* time units at once. No process waits more than (*n*-1)*q* time units.
- Performance
  – *q* large $\Rightarrow$ FIFO
  – *q* small $\Rightarrow$ *q* must be large with respect to context switch, otherwise overhead is too high.



RR Time Quantum


Process allocated time quantum
Interaction complete
Time
Response time s
q - s
Quantum q
Time quantum greater than typical interaction
Process allocated time quantum
Process preempted
Process allocated time quantum
Interaction complete
Time quantum less than typical interaction
q
Other processes run
s



Example of RR with Time Quantum = 20

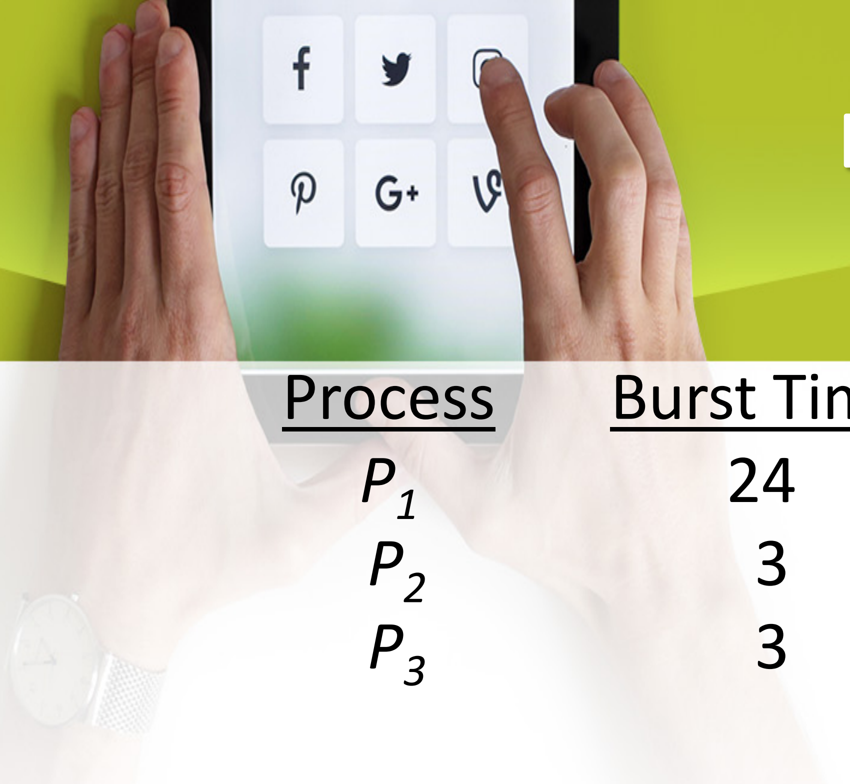
Process	Burst Time
P_1	53
P_2	17
P_3	68
P_4	24

- The Gantt chart is:

P_1	P_2	P_3	P_4	P_1	P_3	P_4	P_1	P_3	P_3

0 20 37 57 77 97 117 121 134 154 162

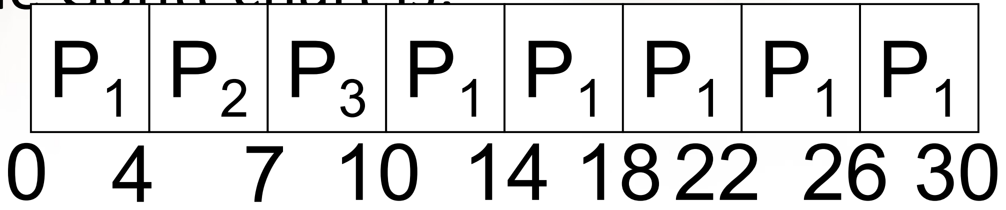
- Typically, higher average turnaround than SJF, but better *response*.

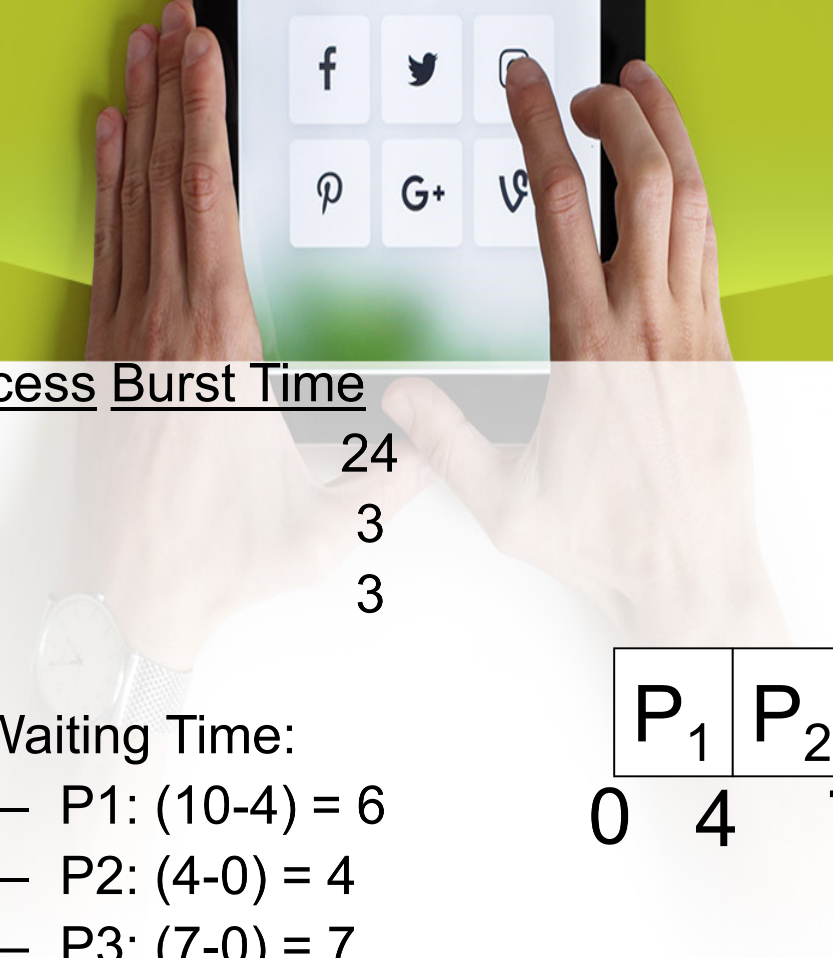


Example of RR with Time Quantum = 4

Process	Burst Time
P_1	24
P_2	3
P_3	3

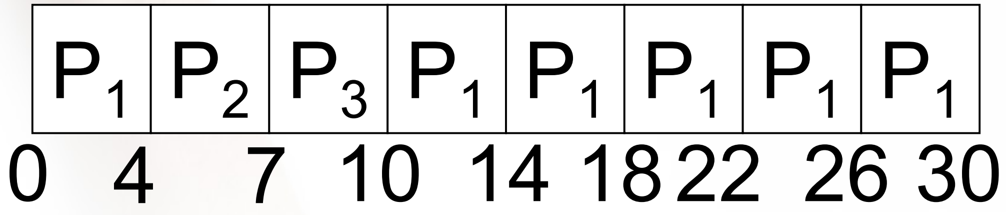
- The Gantt chart is:





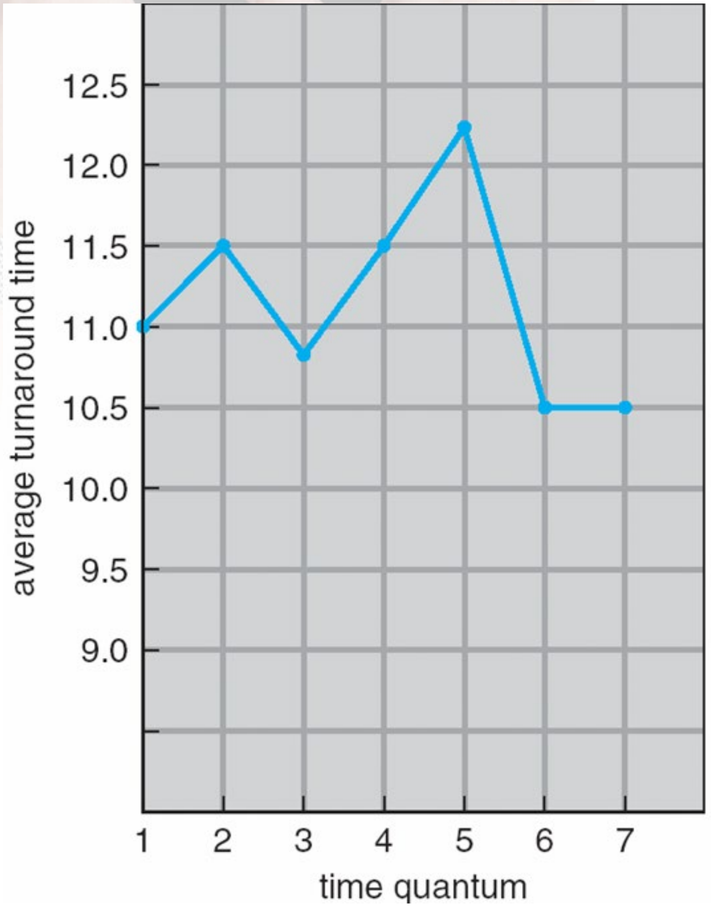
Example of RR with Time Quantum = 4

Process	Burst Time
P_1	24
P_2	3
P_3	3

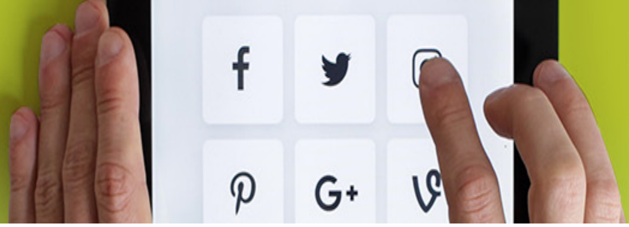


- Waiting Time:
  - P1: (10-4) = 6
  - P2: (4-0) = 4
  - P3: (7-0) = 7
- Completion Time:
  - P1: 30
  - P2: 7
  - P3: 10
- Average Waiting Time: (6 + 4 + 7)/3= 5.67
- Average Completion Time: (30+7+10)/3=15.67

Turnaround Time Varies With The Time Quantum

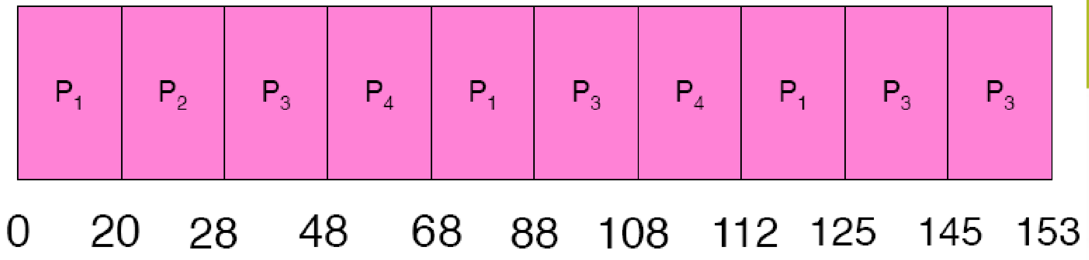


process	time
P_1	6
P_2	3
P_3	1
P_4	7



Quantum = 20

Process	Burst Time
P_1	53
P_2	8
P_3	68
P_4	24



A process can finish before the time quantum expires, and release the CPU.

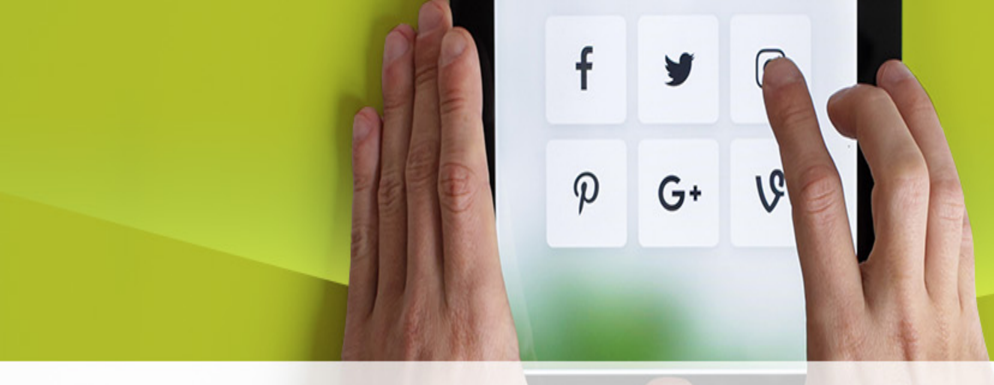
- Waiting Time:
  - P1: (68-20)+(112-88) = 72
  - P2: (20-0) = 20
  - P3: (28-0)+(88-48)+(125-108) = 85
  - P4: (48-0)+(108-68) = 88
- Completion Time:
  - P1: 125
  - P2: 28
  - P3: 153
  - P4: 112
- Average Waiting Time: (72+20+85+88)/4 = 66.25
- Average Completion Time: (125+28+153+112)/4 = 104.5

Weighted Round-Robin

- Weighted round-robin
  - Different weight w_i (per flow)
  - Flow j can sends w_j packets in a period.
  - Period of length Σw_j
- Disadvantage
  - Variable packet size.
  - Fair only over time scales longer than a period time.
    - If a connection has a small weight, or the number of connections is large, this may lead to long periods of unfairness.

DRR (Deficit RR) algorithm

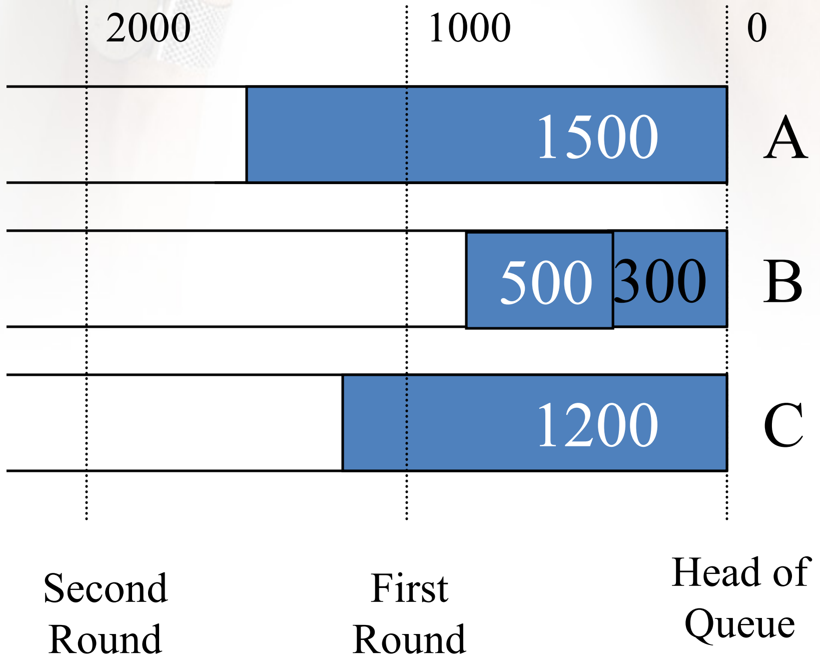
- Choose a quantum of bits to serve from each connection in order.
- For each HoL (Head of Line) packet,
  - credit := credit + quantum
  - if the packet size is ≤ credit; send and save excess,
  - otherwise save entire credit.
  - If no packet to send, reset counter (to remain fair)
- Each connection has a deficit counter (to store credits) with initial value zero.
- Easier implementation than other fair policies
  - WFQ



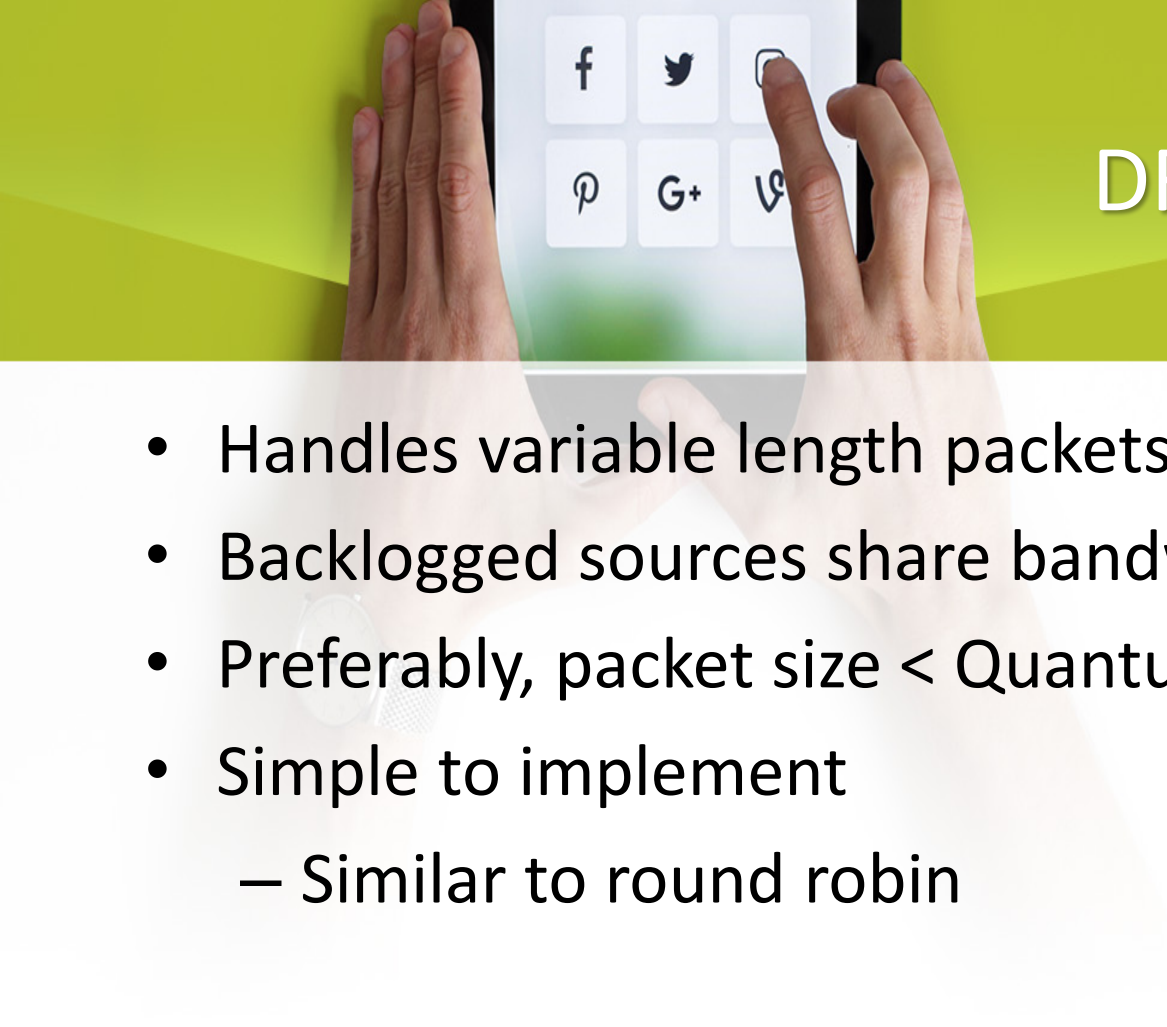
Deficit Round-Robin

- DRR can handle variable packet size

Quantum size : 1000 byte

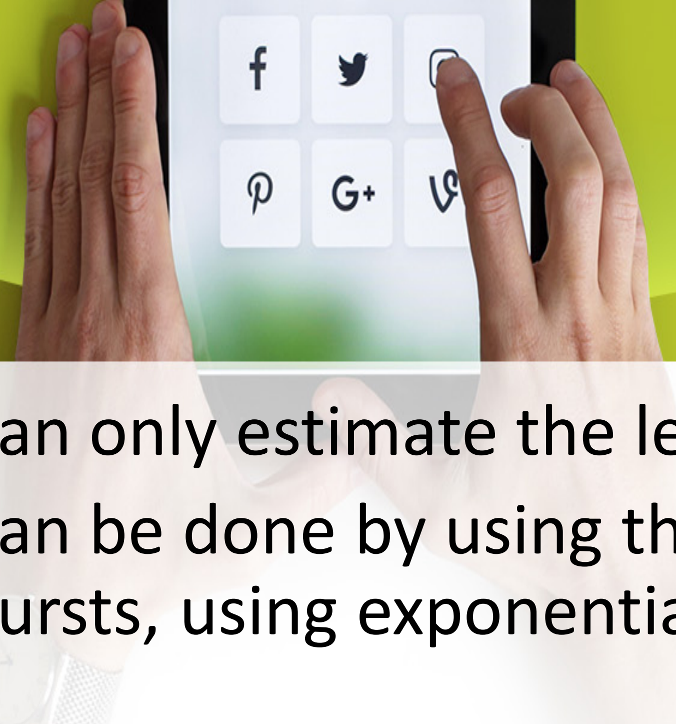


- 1st Round
  - A's count : 1000
  - B's count : 200 (served twice)
  - C's count : 1000
- 2nd Round
  - A's count : 500 (served)
  - B's count : 0
  - C's count : 800 (served)



DRR: performance

- Handles variable length packets fairly
- Backlogged sources share bandwidth equally
- Preferably, packet size < Quantum
- Simple to implement
  - Similar to round robin

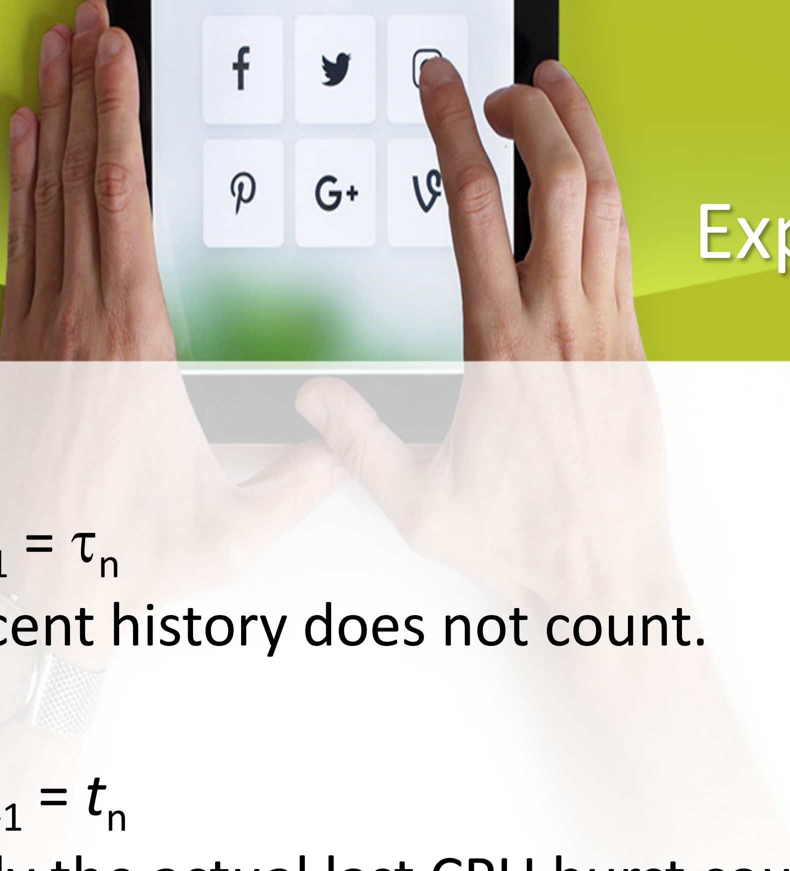


Determining Length of Next CPU Burst

- Can only estimate the length.
- Can be done by using the length of previous CPU bursts, using exponential averaging.

1. $t_n =$ actual lenght of n^{th} CPU burst
2. $\tau_{n+1} =$ predicted value for the next CPU burst
3. $\alpha, 0 \leq \alpha \leq 1$
4. Define :

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



Examples of Exponential Averaging

- $\alpha = 0$
  - $\tau_{n+1} = \tau_n$
  - Recent history does not count.
- $\alpha = 1$
  - $\tau_{n+1} = t_n$
  - Only the actual last CPU burst counts.
- If we expand the formula, we get:

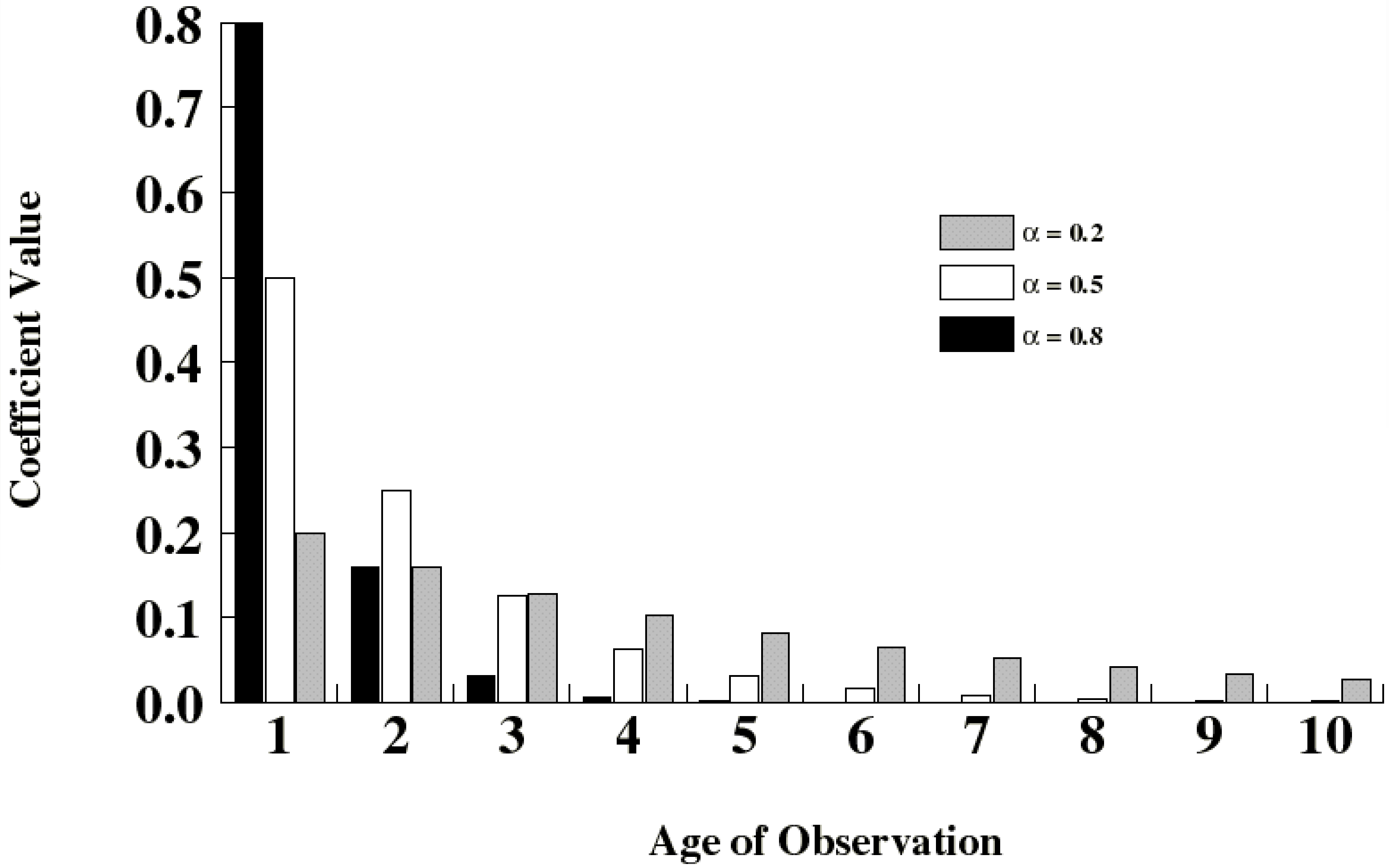
$$\tau_{n+1} = \alpha\, t_n + (1 - \alpha)\, \alpha\, t_{n-1} + \ldots$$
$$+ (1 - \alpha)^j\, \alpha\, t_{n-j} + \ldots$$
$$+ (1 - \alpha)^{n+1}\, \tau_1$$

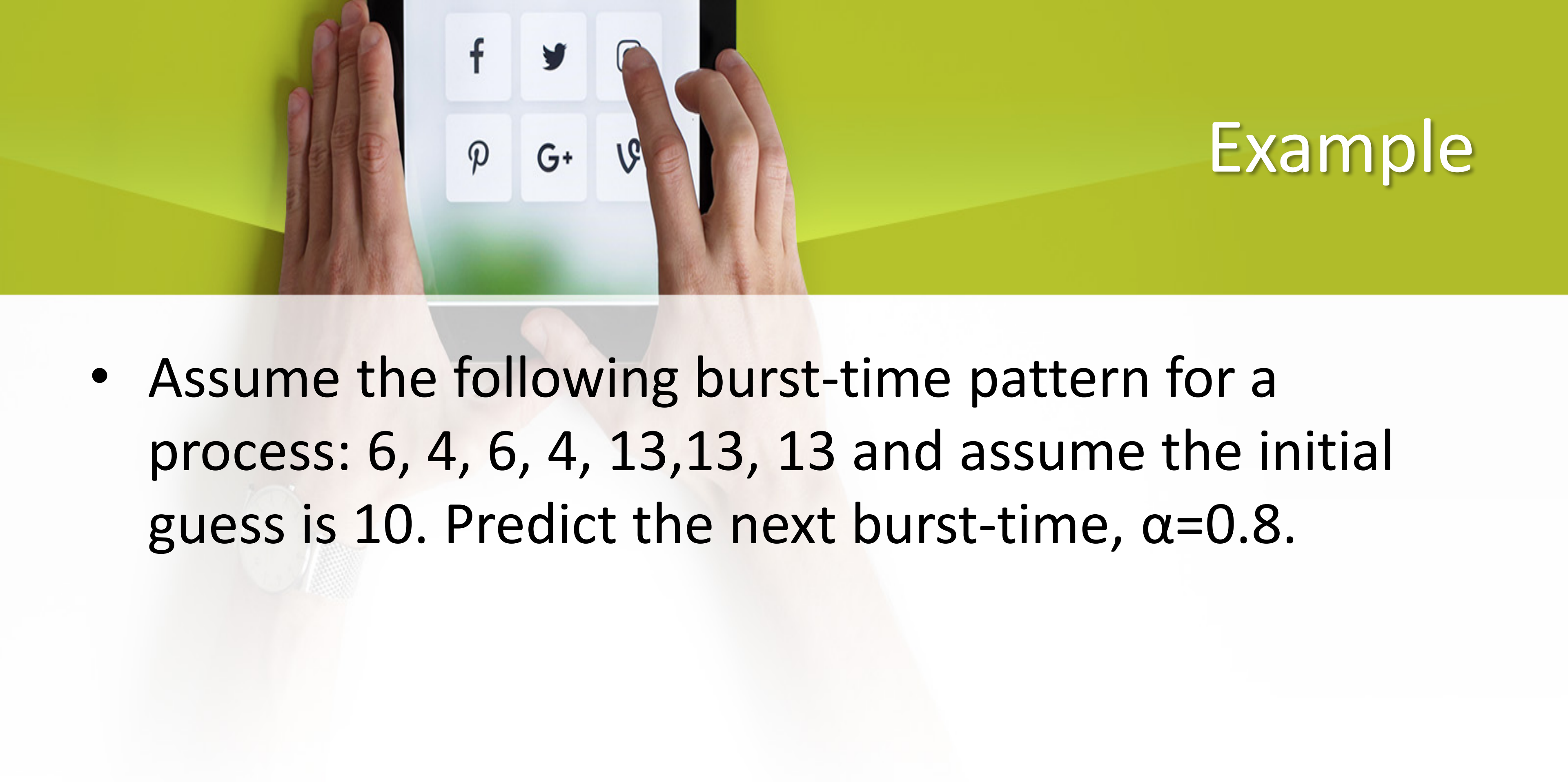
- Since both α and $(1 - \alpha)$ are less than or equal to 1, each successive term has less weight than its predecessor.

More on Exponential Averaging

1. S[n+1] next burst, S[n] current burst (predicted), T[n] actual,
   - $S[n+1] = \alpha T[n] + (1-\alpha) S[n]$; $0 < \alpha < 1$
   - more weight is put on recent instances whenever $\alpha > 1/n$
2. By expanding this eqn, we see that weights of past instances are decreasing exponentially
   - $S[n+1] = \alpha T[n] + (1-\alpha)\alpha T[n-1] + \ldots (1-\alpha)^i \alpha T[n-i] + \ldots + (1-\alpha)^n S[1]$
   - predicted value of 1st instance S[1] is not calculated; usually set to 0 to give priority to new processes

Exponentially Decreasing Coefficients





Example

- Assume the following burst-time pattern for a process: 6, 4, 6, 4, 13,13, 13 and assume the initial guess is 10. Predict the next burst-time, α=0.8.

Sn	10	6.8	4.56	5.71	4.34	11.27	12.49		
Tn	6	4	6	4	13	13	13		
Sn+1	6.8	4.56	5.71	4.34	11.27	12.49	12.89		

Example

- Assume the following burst-time pattern for a process: 6, 4, 6, 4, 13,13, 13 and assume the initial guess is 10. Predict the next burst-time, α=0.2 and compare with α=0.8; try it for α=0.5, 1.0

Sn		10	6.8	4.56	5.71	4.34	11.27	12.49		
Tn		6	4	6	4	13	13	13		
Sn+1	α=0.8	6.8	4.56	5.71	4.34	11.27	12.49	12.89		
Sn+1	α=0.2	8.96	7.808	7.206	6.405	7.204	7.843	8.354		