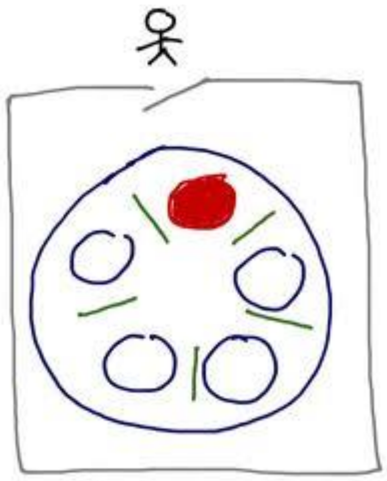
Resource Sharing & Inter-Process Interactions

- **The Issues**: Critical Sections, Mutual Exclusion, Deadlocks…
- ❑ **Task Orderings**: Scheduling issues and solutions
- **Algorithmic** Solutions: Races, Ordering, Alternations…
- **Program Level** Solutions: Semaphores, Monitors

Alternatives?

- Algorithms to allocate a resource
  - E.g., let's give resources to shortest job first
    - Works great for multiple short jobs in a system
    - May cause indefinite postponement of long jobs even though not blocked (starvation?)



- Other Solutions?
  - First-come, first-served (FIFO) policy
  - Shortest job first, highest priority, deadline first etc…

Scheduling Policies!

… efficient, fair, deadlock & race-free

OS Scheduling Criteria (Optimizations)

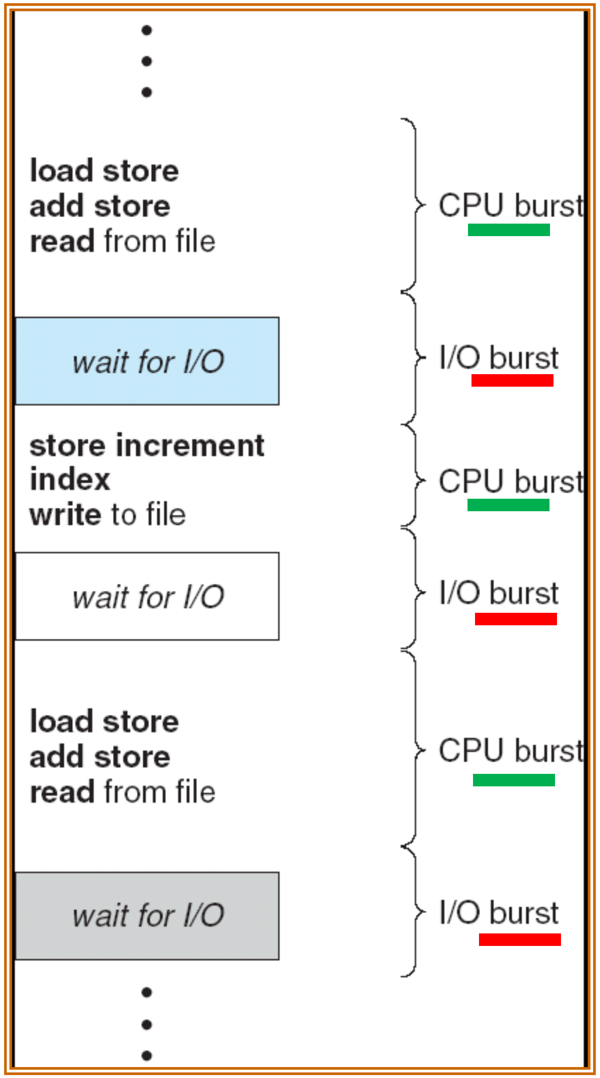
- **CPU utilization:** keep CPU as busy as possible **(maximize)**
  - Nice target but the CPU better be doing something useful! Remember livelock?
- **Throughput:** # of processes completed/time unit **(maximize)**
- **Turnaround time:** average amount of time to execute a particular process **(minimize)**
- **Waiting time:** process wait time in ready queue **(minimize)**
- **Response time (minimize):** amount of time it takes from when a request was submitted until the first response (**not** output) is produced. [result output time = RT deadline]

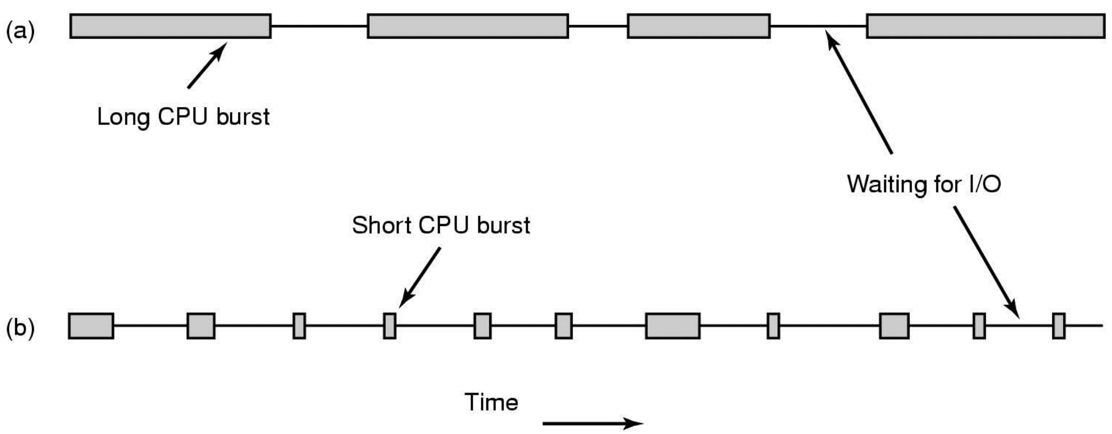
➢ **Efficiency ➔ Optimize all ☺ …plus want Fairness! …and across very diverse applications!**

Scheduling Variants

A. First Come, First Served (**FCFS**)
B. Shortest Job First (**SJF**)
C. Round Robin (**RR**)
D. Priority Based (**PB**)

Issue 1. No Single Process/CPU Usage Model

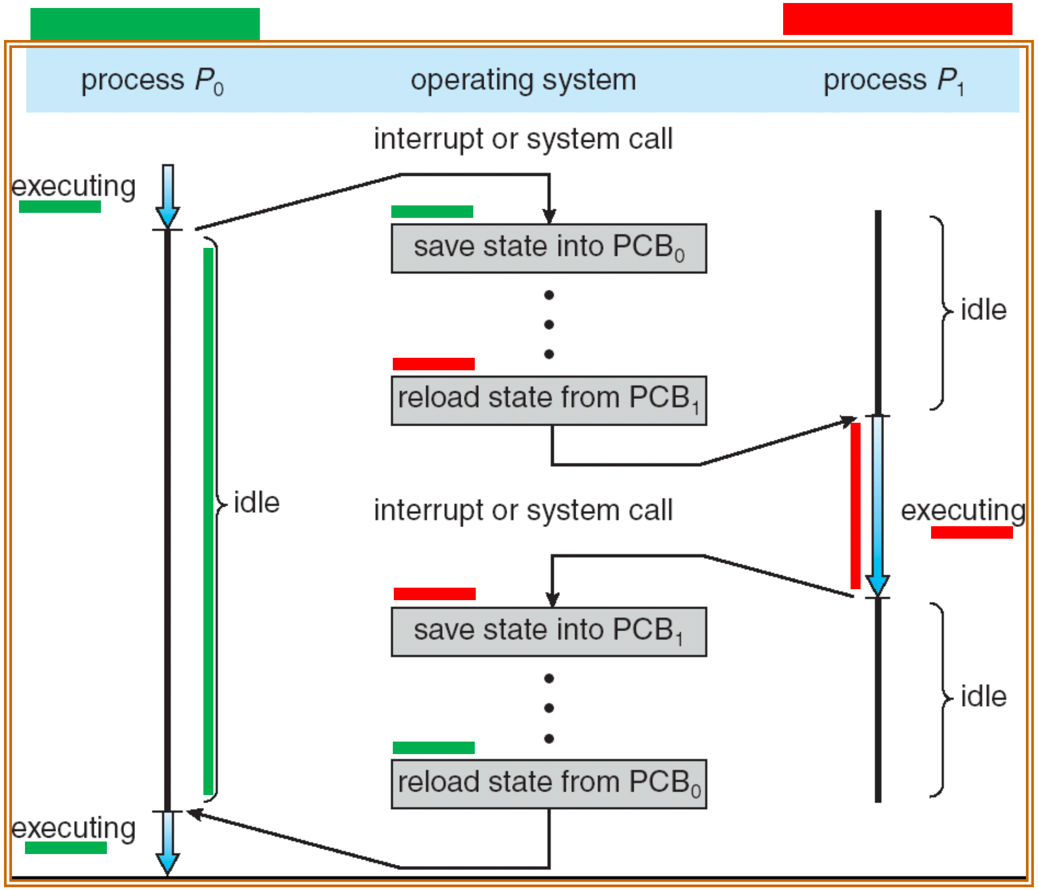




- Apps “typically” stay CPU or I/O-bound
- Hybrids “typically” for interactive apps

Issue 2. Process/Thread Premptability

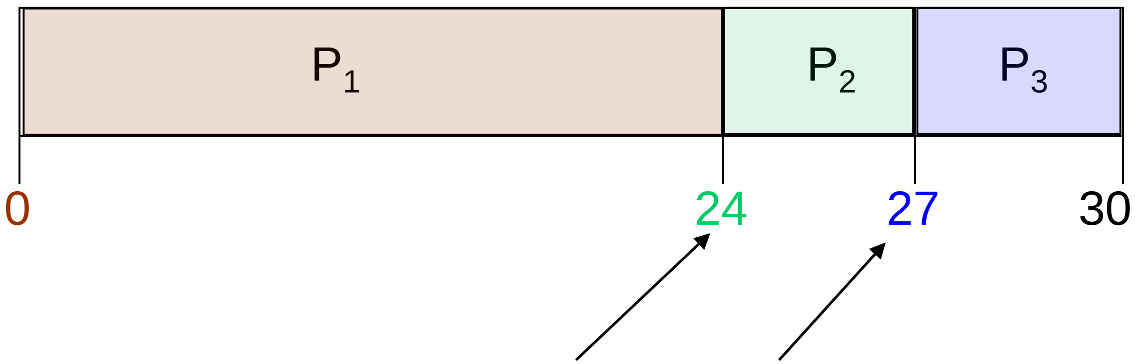
- **Non preemptive (NP)**: An ongoing task cannot be displaced
- **Preemptive**: Ongoing tasks can be switched in/out as needed



A: First-Come, First-Served (FCFS) Scheduling **(Non-Preemptive)**

Process	Execution Length (CPU Burst Time)
P_1	24
P_2	3
P_3	3

- Processes arrive (& get executed) in their arrival order: $\boldsymbol{P_1}$ **,** $\boldsymbol{P_2}$ **,** $\boldsymbol{P_3}$
  Single queue of "ready" non-premptible processes



- Waiting time for P_1 = 0; $\boldsymbol{P_2}$ **= 24**; P_3 = 27
- Average waiting time: (0 + 24 + 27)/3 = **17**

FCFS Scheduling **(NP: Non-Preemptive)**

Suppose that the processes arrive in a different order as:

$\boldsymbol{P_2 , P_3 , P_1}$ [3,3,24]

- Process Order/Schedule:

P_2	P_3	P_1

0 3 6 30

- Waiting time for $P_1 = 6$; $P_2 = 0$; $P_3 = 3$
- Average waiting time: $(6 + 0 + 3)/3 =$ **3**

- Variability of wait time (from 17 to 3!) – control? response prediction?

Scheduling in Batch Systems where we can actually see and re-order FCFS?

Shortest job first (SJF) scheduling?

B: Shortest-Job-First (SJF) Scheduling

- Associate with each process the length of expected CPU burst
- Use these lengths to schedule the process with the shortest time

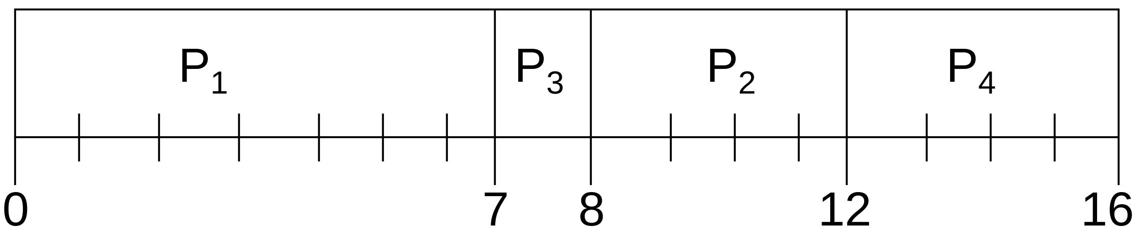
Options:

- non-preemptive – once CPU given to the process it cannot be preempted until it completes its CPU burst
- preemptive – if a new process arrives with CPU burst length less than remaining time of current executing process, preempt.
  - Variation: Shortest-Remaining-Time-First (SRTF)

Example of **Non-Preemptive** SJF

Process	Arrival Time		Burst Time
P_1	0.00	0.0	7
P_2	0.0+	2.0	4
P_3	0.0+	4.0	1
P_4	0.0+	5.0	4

- SJF (non-preemptive) P_1, then P_3 then P_2, P_4 (FCFS across equals)



- Average waiting time = (0 + 8 + 7 + 12)/4 = **6.75** (fixed arrival)
- Av. waiting = (0 + [8-2] + [7-4] + [12-5])/4 = **4.00** (staggered arrival)

Shortest-Job-First (SJF) Scheduling

- Associate with each process the length of its next CPU burst
- Use these lengths to schedule the process with the shortest time

Options:

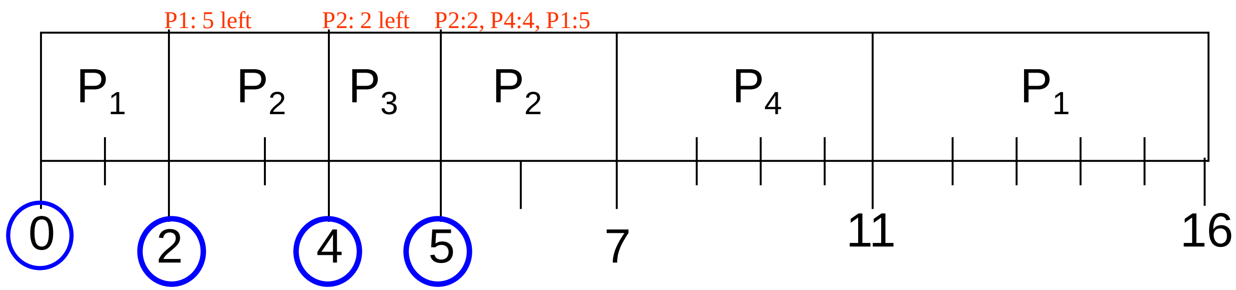
- non-preemptive – once CPU given to the process it cannot be preempted until it completes its CPU burst
- preemptive – if a new process arrives with CPU burst length less than remaining time of current executing process, preempt.
  Variation: Shortest-Remaining-Time-First (SRTF)

Preemptive SJF

(Shortest Remaining Time First: Re-assess SRT on arrivals)

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

- SJF (preemptive)



- Average waiting time = P1:[0, (11-2)]; P2:[0, (5-4)]; P3: 0; P4: (7-5)
- Average waiting time = (9 + 1 + 0 +2)/4 = **3** …[6.75;4.00]
- Dynamic scheduling (SRT) calculations + context switches ➔ **cost !!!**

Determining Length of Next CPU Burst

- Can only estimate the task length (simple for batch jobs)
- Use the length of previous CPU bursts (history logs & ageing)

1. t_n = actual length of n^{th} CPU burst
2. τ_{n+1} = predicted value for the next CPU burst
3. α, $0 \le \alpha \le 1$;
4. Define :

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

$$\tau_{n+1} = \tau_n$$ (α = 0) direct fn. of prior prediction

$$\tau_{n+1} = t_n$$ (α = 1) direct fn. of most recent run

α = 1/2

$\tau_1 = ½\, t_o + ½\, \tau_0$ add + simple right shift (→divide by 2)

$\tau_2 = ½\, t_1 + ½\, \tau_1 = ½\, t_1 + ½\,(½\, t_0 + ½\, \tau_0) = ½\, t_1 + ¼\, t_0 + ¼\, \tau_0$

$\tau_3 = ½\, t_2 + ½\, \tau_2 = ½\, t_2 + ¼\, t_1 + 1/8\, t_0 + 1/8\, \tau_0$

Shortest-Job-First (SJF) Scheduling

- Associate with each process the length of its next CPU burst
- Use these lengths to schedule the process with the shortest time

  Options:
  - non-preemptive – once CPU given to the process it cannot be preempted until completes its CPU burst
  - preemptive – if a new process arrives with CPU burst length less than remaining time of current executing process, preempt. Shortest-Remaining-Time-First (SRTF)

- SJF is optimal – gives minimum average waiting time for a given set of processes
- SJF can still lead to starvation: if new short jobs keep coming in and getting considered in the "shortest" ranking, a "long" job can potentially starve! [Convoy Effect]

Variations

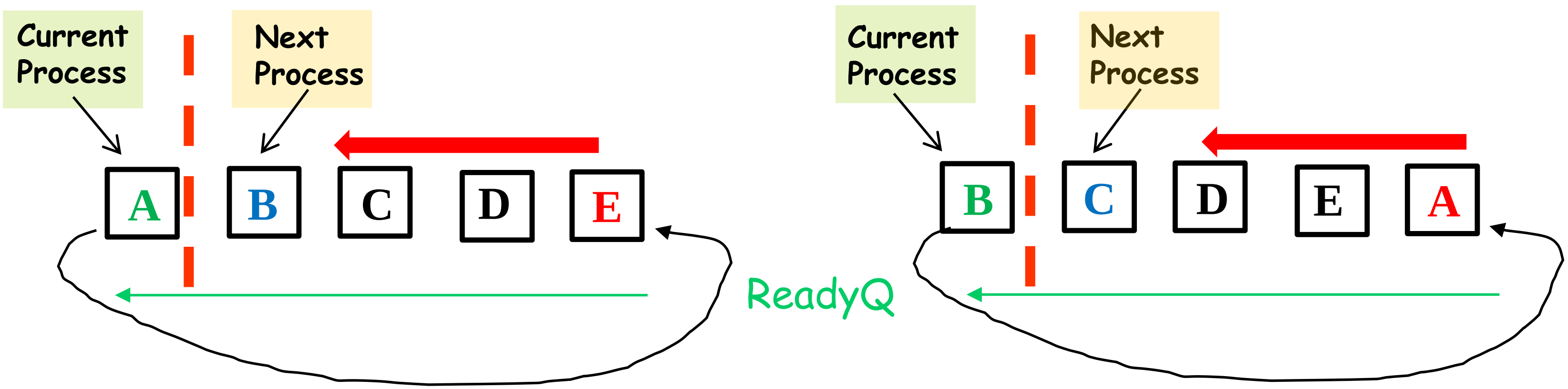
- ✓ SJF ➔ EDF (Earliest Deadline First)

Onwards to Preemptive Scheduling

Preemptive FCFS → Round Robin

C: *Round Robin (RR/TDMA): Fixed Slots

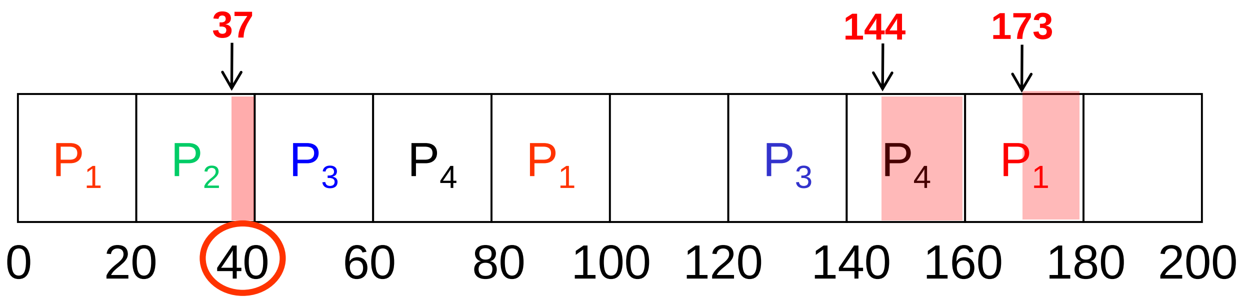
- Serial ReadyQ of processes (Circular Queue)
- Each process gets a fixed slice of CPU time *(time quantum: q)*, 10-100ms
  - Slot finishes ➔ process "**preempted**", added to the end of Q: **Sliding Window**
  - *n* processes in ReadyQ and time quantum is *q* → each process gets 1/*n* of CPU time in chunks of *q* time units. No process waits more than (*n*-1)*q* time units.



- FIFO Performance?
  - *q* too large ⇒ poor response time & poor CPU utilization – wasted time
  - *q* too small ⇒ context switching. *q* must be large wrt context switch overhead, else context switching overhead reduces efficiency

RR with Fixed/Static time Quantum = 20

Process	Burst Time
P_1	53
P_2	17
P_3	68
P_4	24



Better: Dynamic RR with Quantum = 20

Dynamic Q: if a process finishes in less than 20, then start the next slot earlier!

Process	Burst Time
P_1	53
P_2	17
P_3	68
P_4	24

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*
  - **Turnaround time** – amount of time to execute a particular process (minimize)
  - **Response time** (min) – amount of time it takes from request submission until first response
- Scheduling is static though calculating/estimating “right” length of quantum is crucial!!!
- Alternate: Have quanta of varying lengths!!! (dynamic scheduling costs ~ SJF)

Input Queue Policy?

- **Static** – Preemptive FCFS scheduling with predictable wait time
- **Dynamic?**
  - If Q keeps adding new jobs → slot waiting times can vary (extend!) for existing jobs leading to potential starvation (Admission Control)

D: *Priority Scheduling

- A priority number (integer) is associated with each process
- The CPU is <u>allocated</u> to the process with the <u>highest</u> priority

  Generally, **Highest =‘s Biggest #** … BUT in Unix: smallest integer ≡ highest priority

  – Preemptive (priority and order changes possible at runtime)

  – Non-preemptive (initial execution order chosen by static priorities)

> - FCFS is priority scheduling where arrival order determines priority
> - SJF is priority scheduling where CPU burst time determines priority
> - RR is equal priority scheduling (albeit in FCFS order)

- <u>Problem</u> ≡ Starvation: low priority processes may never execute
- <u>Solution</u> ≡ **Ageing**: as time progresses increase priority of process

Ageing and Priority

- Priority can be static or dynamic (as a function of wait-time & usage-time for fairness etc: Unix BSD, Linux)

  Note: In BSD ➔ Large # =‘s Low Priority!

P_user-pri = P_USER + P_cpu/4 + (2 * P_nice)

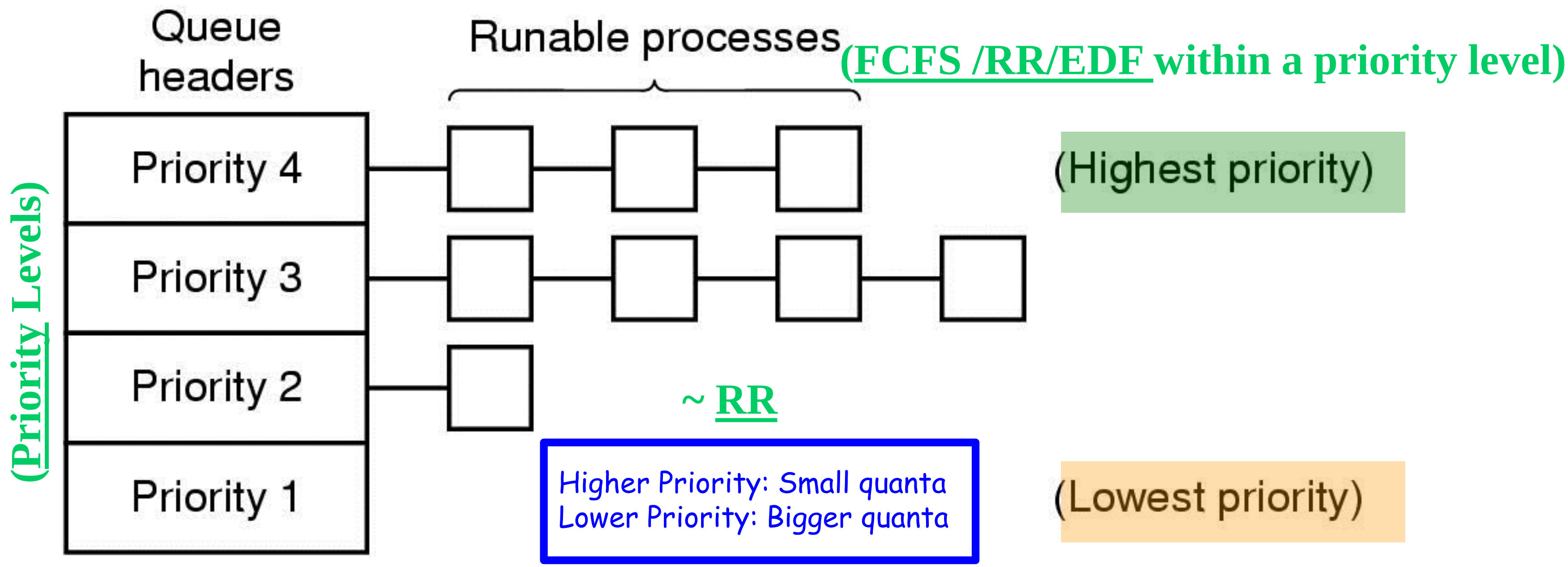
- Process **using** CPU: **P_cpu increases** with time ➔ **P_user-pri goes up** (ie lower priority)
- Process **waiting**: **P_cpu** keeps **decreasing** ➔ **P_user-pri** keeps going **down** (ie higher priority)

Which Scheduling Policy to Use?

- First Come, First Served (**FCFS**)
- Shortest Job First (**SJF**)
- Round Robin (**RR**)
- Priority Based (**PB**)

- OS schedulers implement *<u>all</u>* scheduling policies
- Choice of policy is based on the nature of specific applications (computing, I/O, multimedia etc)
- Mixture of applications (at kernel- level, at user-level) determines the specific scheduling policy or groups of policies to use!

Scheduling Options (Solaris, Windows, Linux)

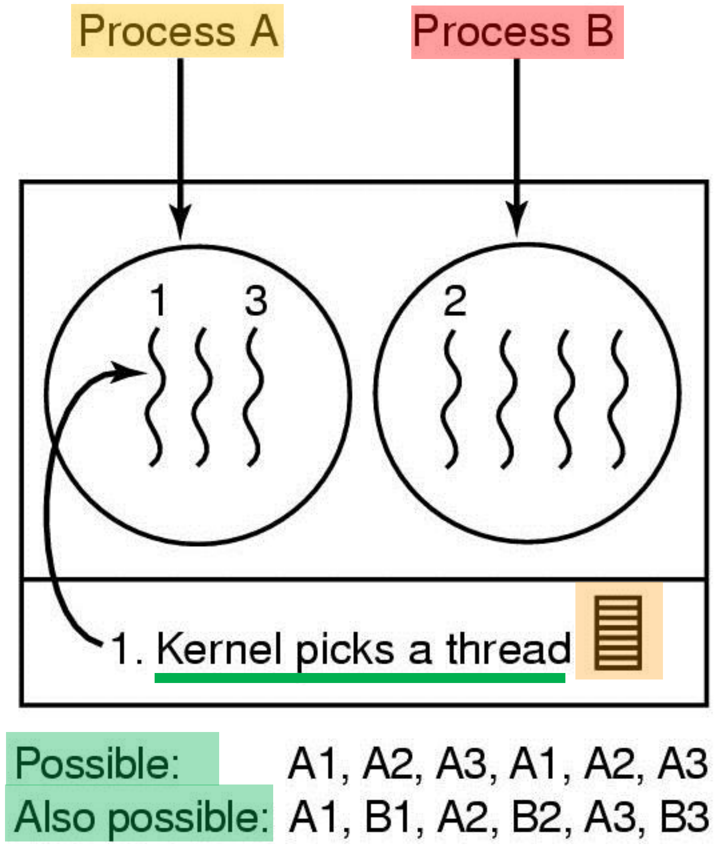


- **Kernel process**: "generally" non-preemptive
- User processes: "mostly" preemptive

Where Should the Scheduler be?

- ❑ Kernel level?
- ❑ User level?

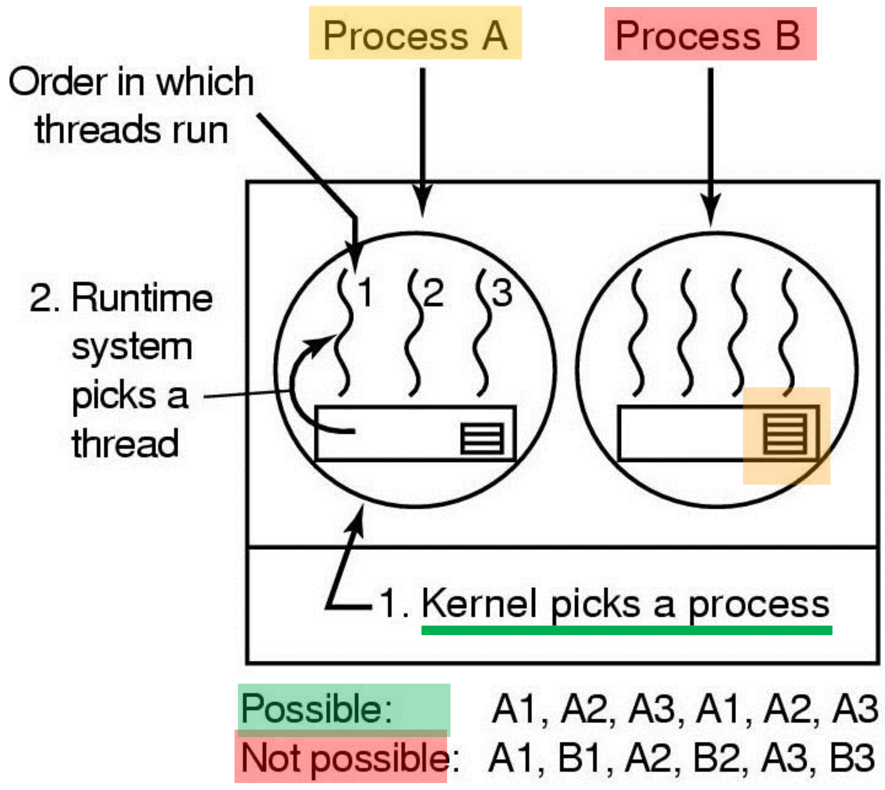
Thread Scheduling (Kernel Scheduler)



Possible scheduling of kernel-level threads

- Flexible & efficient for quanta usage…but costly full context switch for threads
- Monolithic kernel scheduler also avoids thread blocks (on I/O etc)

Thread Scheduling (<u>User Scheduler</u>)

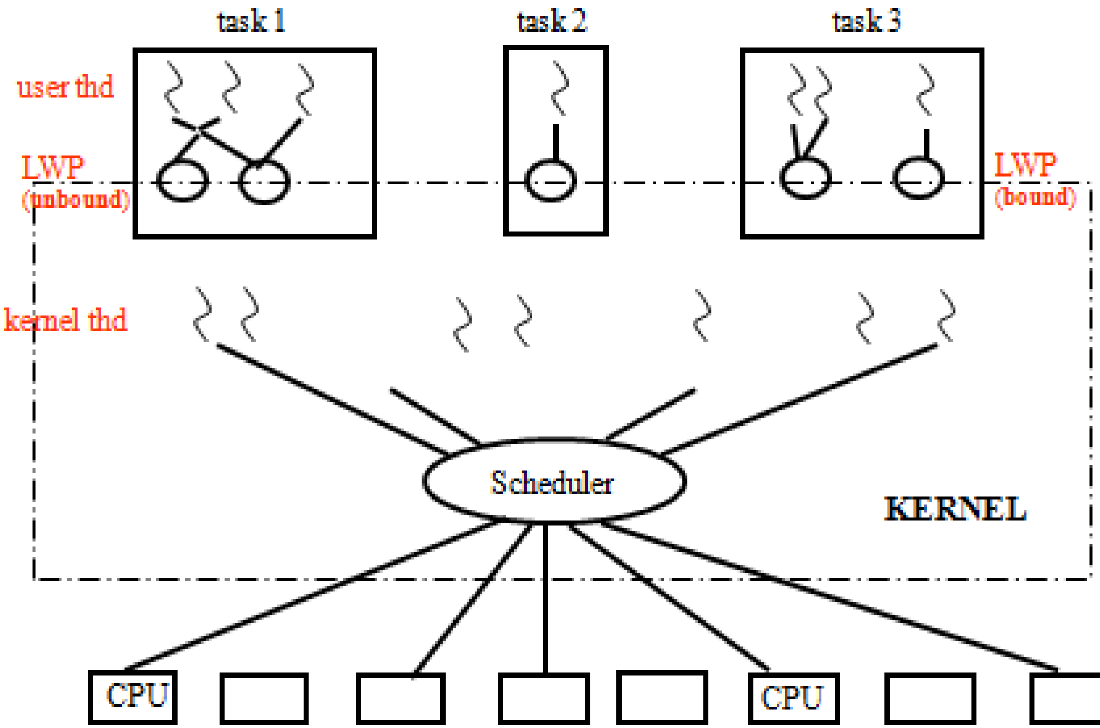


Possible scheduling of user-level threads

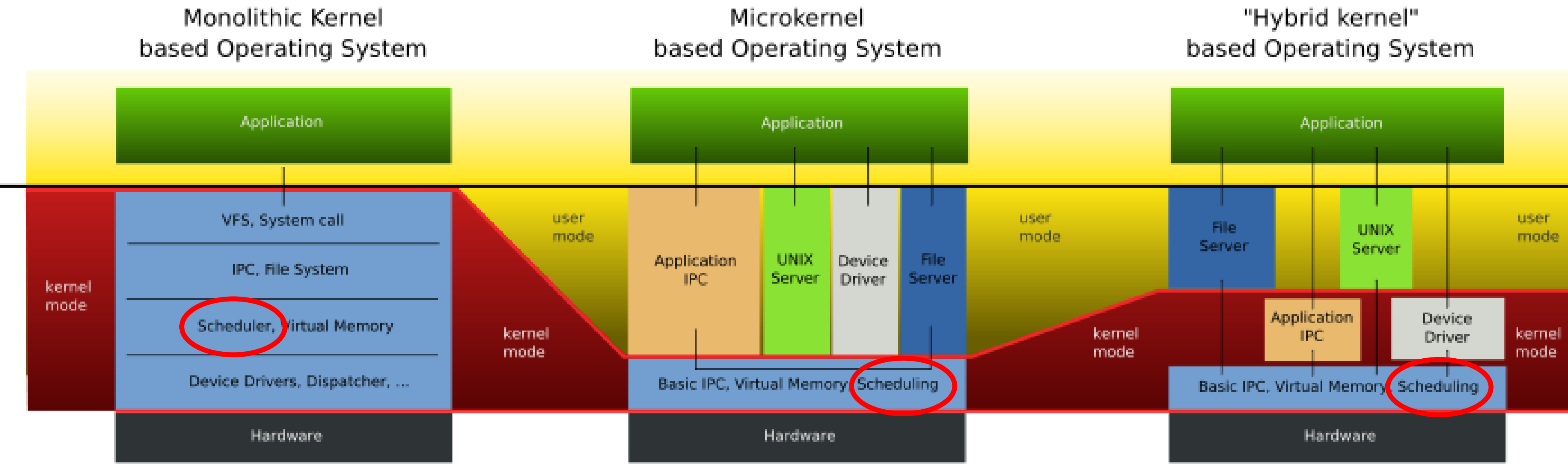
- <u>Simple</u> process level context switch; thread block → process blocks
- <u>Split level</u> scheduling (kernel + dedicated application/thread type specific)

Thread Scheduling

- <u>Local Scheduling</u>: Threads library decides which thread to put onto an available LWP
- <u>Global Scheduling</u>: Kernel decides which kernel thread to run next



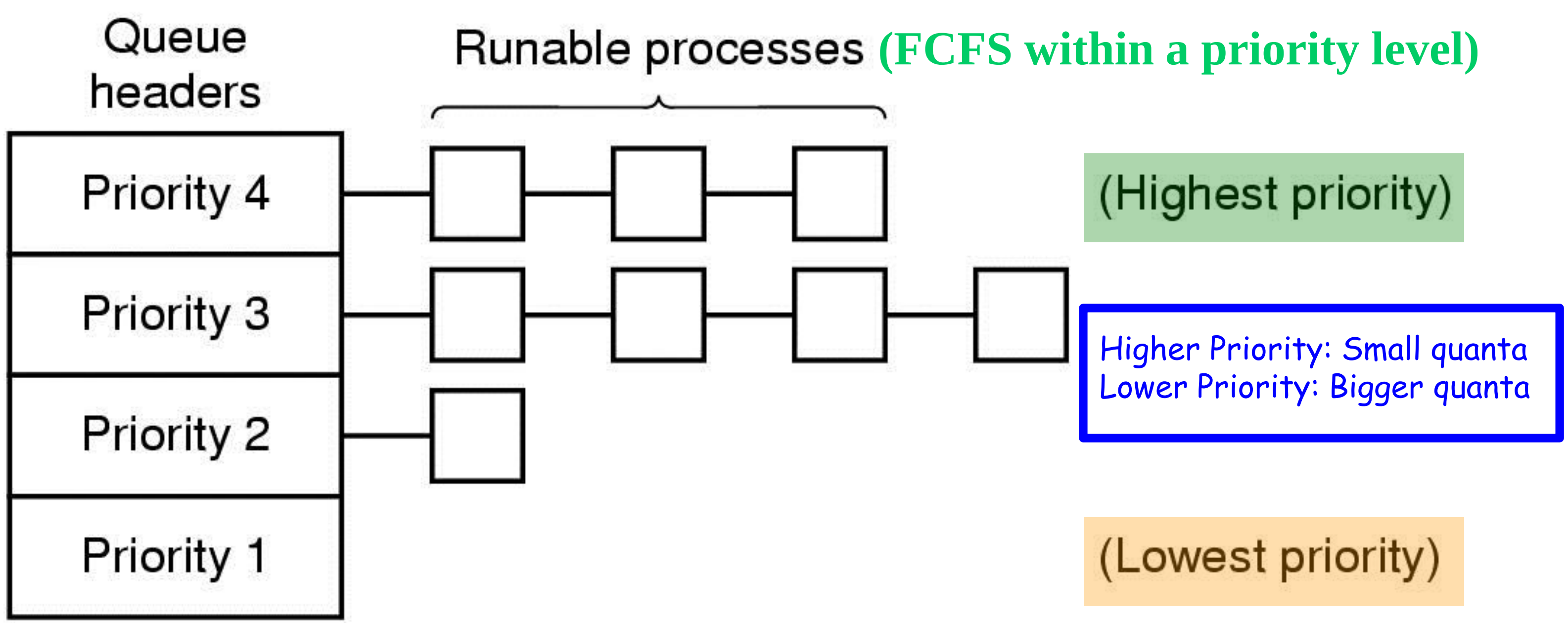
Scheduler Locations?



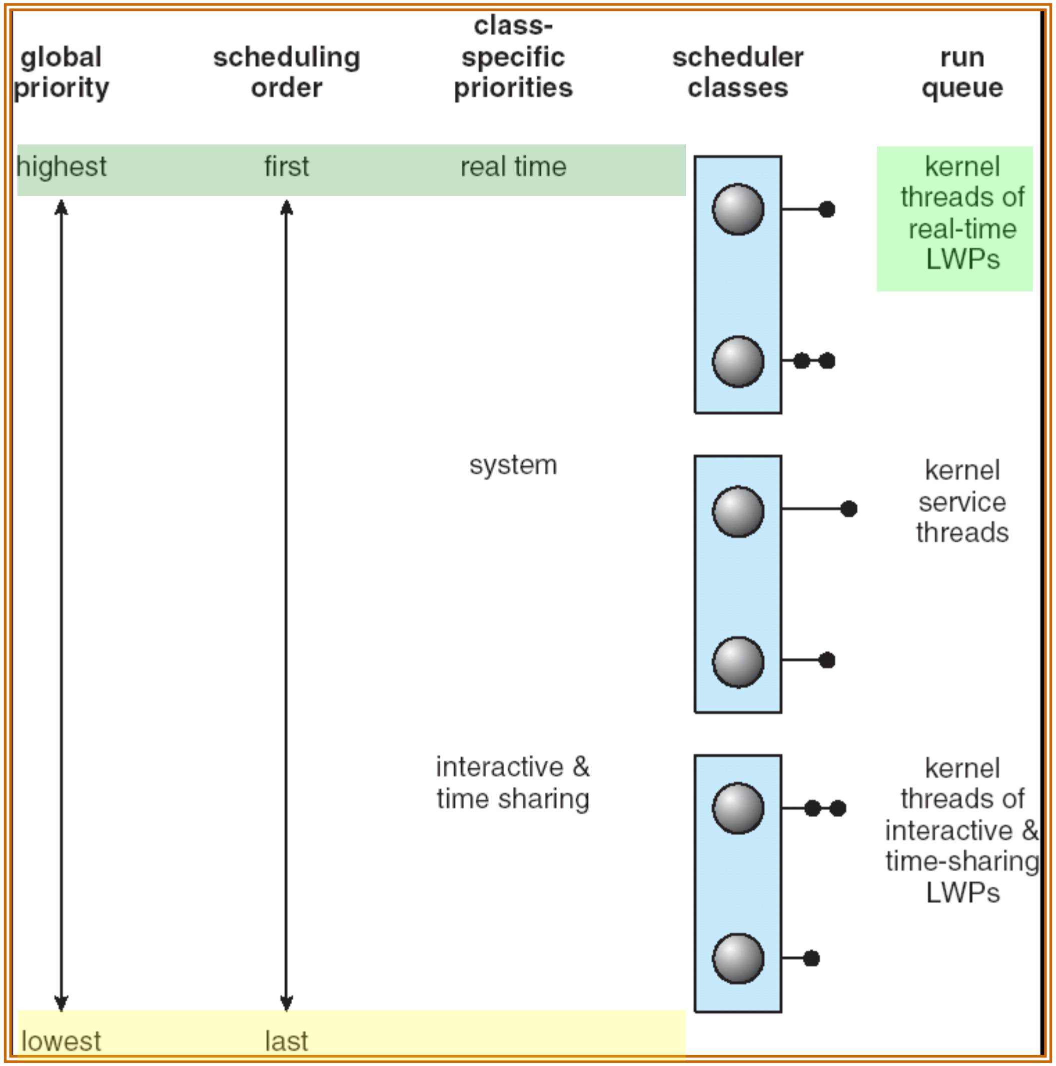
OS Examples

- ❑ Solaris scheduling
- ❑ Windows scheduling
- ❑ Linux scheduling

- All work with dynamically altering priorities & time slices
- All have preemptive tasking



Linux/Solaris Scheduling



Higher the priority, shorter the time slice

Solaris Dispatch Table

TQE RFS ← New dynamically altered priorities …and time slots

priority	time quantum(msec)	time quantum expired	return from sleep
0	200	0	50
5	200	0	50
10	160	0	51
15	160	5	51
20	120	10	52
25	120	15	52
30	80	20	53
35	80	25	54
40	40	30	55
45	40	35	56

CPU-tasks mid-priority,
Interactive: higher priority

* Lowest priority 0 (diff. from BSD) is given largest quantum

TQE: amount of quanta used without blocking
(CPU Intensive Tasks) ; Solaris 9+ : No TQE (fixed priority & CPU shares)

RFS: (raised) priority of awakened tasks (responsiveness for I/O Tasks)

Windows: Priority Based, Preemptive

(Fixed & Variable Components)

priority classes

	real-time	high	above normal	normal	below normal	idle priority
time-critical	31	15	15	15	15	15
highest	26	15	12	10	8	6
above normal	25	14	11	9	7	5
normal	24	13	10	8	6	4
below normal	23	12	9	7	5	3
lowest	22	11	8	6	4	2
idle	16	1	1	1	1	1

relative priority **within** a priority class (High # =‘s High Priority; diff. from BSD)

- Fixed: Priority Class (eg RT) will always run when invoked
-Variable: Thread runs till quanta finished; priority lowered; task coming in from “wait”gets priority raised – but only within its priority class!

Linux Scheduling (0=highest)

❖ Two algorithms: time-sharing and real-time

✓ Time-sharing

- Prioritized credit-based – process with most credits is scheduled next
- Credit subtracted when timer interrupt occurs
- When credit = 0, another process chosen
- When all processes have credit = 0, re-crediting occurs
  - Based on factors including priority and history

✓ Real-time

- Soft real-time
- Two classes
  - FCFS and RR
  - Highest priority process always runs first