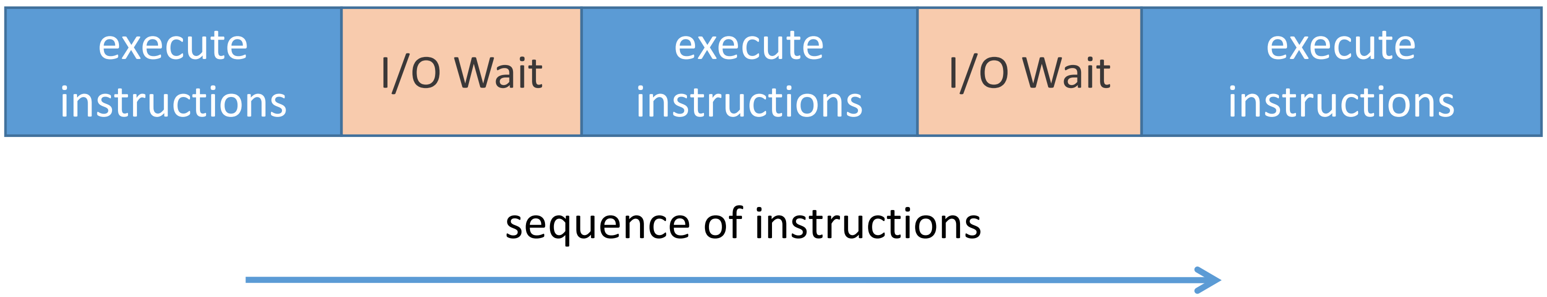
UNIVERSITY of WEST FLORIDA

COP4634: Systems & Networks I

Scheduling

Background

- Alternating sequence of CPU and I/O bursts



- Multiprogramming improves CPU utilization.
  - when one process waits for an I/O device to respond, another process can run on the CPU

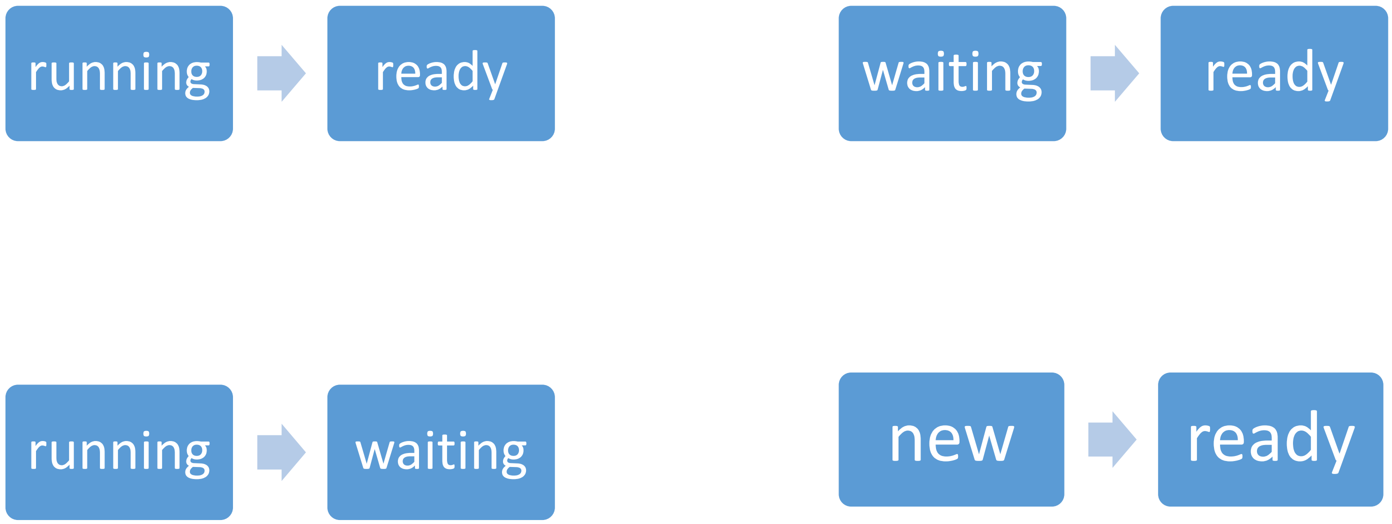
- OS must schedule another process while a process waits for an I/O response

CPU Scheduling Overview

- Allocate CPU to the next process ready to execute.
- Preemptive scheduling means:
  - switching a process from running to ready state
  - switching a process from waiting to ready state
- Preemption requires hardware support.

CPU Scheduling Overview (cont.)

- CPU scheduling is needed when:



Scheduling Components

Dispatcher

Loop

Select process (P_i) from Ready List

*Set timer for now + quantum

Put selected process on CPU to run

… {Alarm – Quantum Expires – Interrupt}

Save state of P_i in PCB

Put P_i in Ready List or Blocked List depending on type of interrupt.

Go to top of Loop for next process

Scheduling Components

Scheduling Algorithm

Loop

Select process (P_i) from Ready List

*Set timer for now + quantum

Put selected process on CPU to run

… {Alarm – Quantum Expires – Interrupt}

Save state of P_i in PCB

Put P_i in Ready List

Go to top of Loop for next process

How?

Where?

Scheduling Goals

- Giving each process a fair share of CPU access
- Ensuring that all policies are properly enforced.
- Keeping all parts of the system equally busy.
- Responding to user requests as quickly as possible.
- Meet user’s expectations as best as possible.
- Real-time systems:
  - meeting scheduling constraints to guarantee timely responsiveness

I/O or Compute?

I/O bound – mostly I/O

CPU bound – little I/O

Schedule which first?

- I/O bound first – Why?
- I/O scheduled, starts I/O, blocked
- When I/O blocked, schedule compute
- Utilize multiple components
- Better average turnaround time

Scheduling I/O First

Ready List

C_2 IO_2 IO_1 IO_0

Blocked List

CPU

Disk

Network

Process IO_0 on CPU

CPU

IO_0

Ready List

C_2 IO_2 IO_1

Disk

Blocked List

Network

IO_0 to Network, IO_1 on CPU

CPU

IO_1

Ready List

C_2 IO_2

Disk

Blocked List

IO_0

Network

IO_0

IO_1 to Disk, IO_2 on CPU

Ready List

C_2

Blocked List

IO_1 IO_0

CPU

IO_2

Disk

IO_1

Network

IO_0

All Devices Busy

CPU

C_2

Ready List

Disk

IO_1

Blocked List

IO_2 IO_1 IO_0

Network

IO_2 IO_0

IO_0 to Ready List

	CPU
	C_2
Ready List	
IO_0	Disk
	IO_1
Blocked List	
IO_2 IO_1	Network
	IO_2

IO_0 to CPU

Ready List

C_2

Blocked List

IO_2 IO_1

CPU

IO_0

Disk

IO_1

Network

IO_2

C_2 back to CPU

Ready List

Blocked List

IO_0 IO_2 IO_1

CPU

C_2

Disk

IO_1

Network

IO_0 IO_2

Optimization Criteria

Goal: Keep CPU as busy as possible.

- Minimize waiting time
  - wait = sum of times process is in ready queue
- Minimize response time: ($t_e - t_a$)
- Turn-around time: ($t_d - t_a$)

Optimization Criteria (cont.)

- Maximize throughput
  - Throughput - # jobs completed per unit of time
  - Throughput ≈ 1 / Turnaround
  - Minimize turnaround →Maximize throughput

- Fairness
  - usually a trade-off

Classes of Resources

Preemptive

- Can take resource at any time
- Control passes back to kernel
- Internal and external events can cause
  - Internal: `yield()`, `exit()`, etc.
  - External: Quantum expiration

Classes of Resources

Non-Preemptive

- Can’t take resource from process
- Process voluntarily gives up resource
- External events not allowed
  - No quantum expirations
- Only internal events can cause
  - System calls: `yield()`, `exit()`
- CPU: “*Run ‘till done*”

FIFO

Non-preemptive

1st on CPU, when done, 2nd on CPU

+ Simple to implement (no switching)

+ Little overhead (no switching)

- Short jobs stuck behind long jobs

Example for all scheduling types

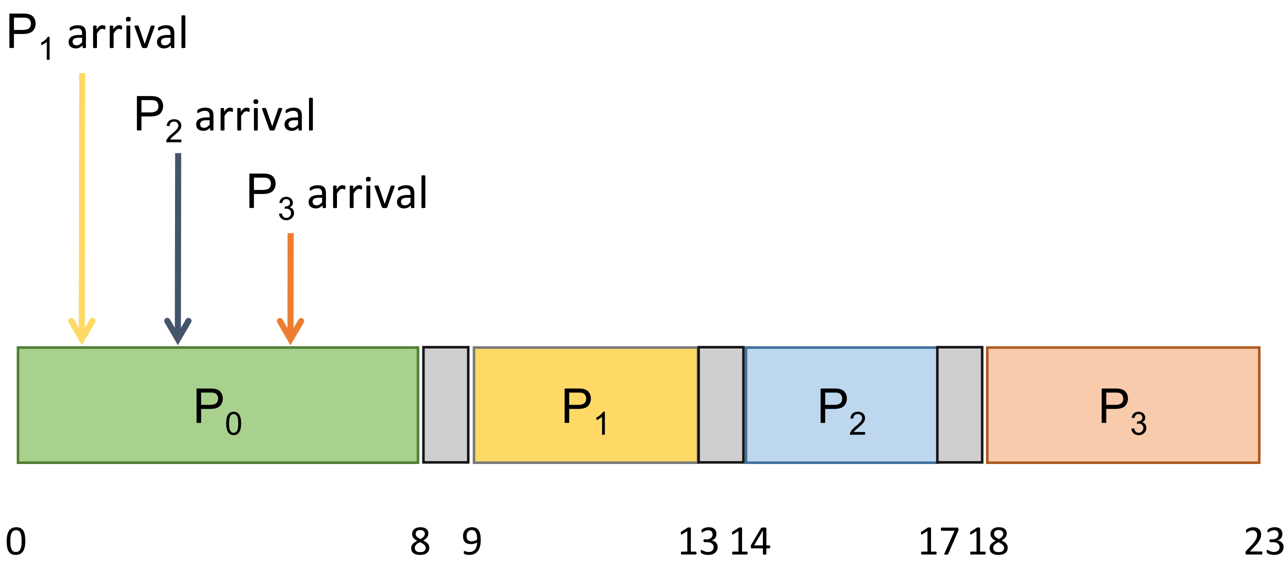
Only a simple example – 4 processes

Example FIFO

Job	t_a	CPU	t_e	t_d	Resp	T/A
P_0	0	8				
P_1	2	4				
P_2	4	3				
P_3	6	5				
				AVERAGE		
				OVERHEAD		

t_a = arrival time
t_e = execution start time
t_d = departure time
Resp = response time
T/A = turnaround time

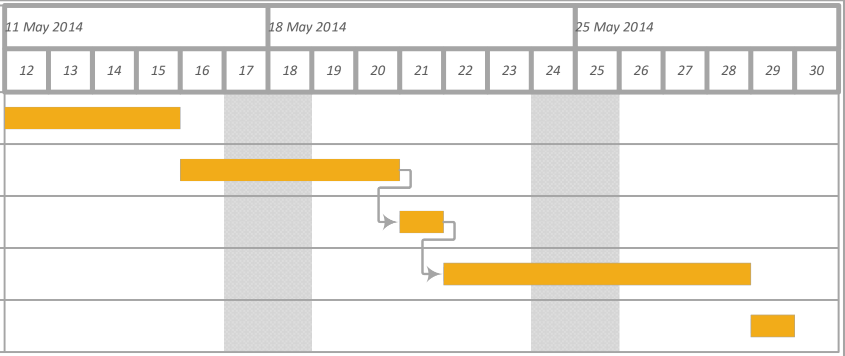
Gantt Chart Illustration



Gantt Chart Examples

Basic Gantt Chart

ID	Task Name	Start	Finish	Duration
1	Task 1	5/12/2014	5/15/2014	4d
2	Task 2	5/16/2014	5/20/2014	3d
3	Task 3	5/21/2014	5/21/2014	1d
4	Task 4	5/22/2014	5/28/2014	5d
5	Task 5	5/29/2014	5/29/2014	1d



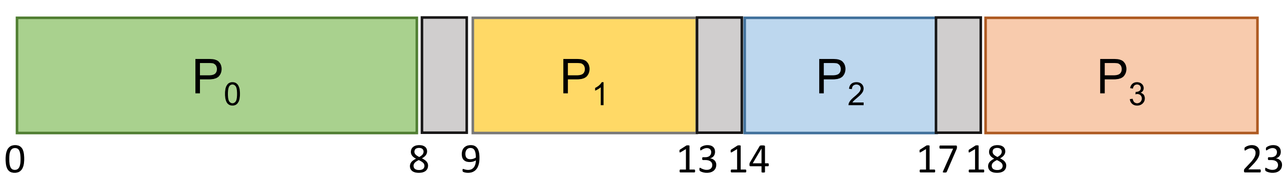
FIFO

0	P_0
8	K
9	P_1
13	K
14	P_2
17	K
18	P_3
23	K

FIFO

Job	t_a	CPU	t_e	t_d	Resp	T/A
P_0	0	8	0	8	0	8
P_1	2	4	9	13	7	11
P_2	4	3	14	17	10	13
P_3	6	5	18	23	12	17
				AVERAGE	7.25	12.25
				OVERHEAD	3	

t_a = arrival time
t_e = execution start time
t_d = departure time
Resp = response time
T/A = turnaround time



Round Robin

Preemptive version of FIFO

Ready list is queue

Quantum used to generate interrupts

- + Relatively simple
- + Fair: each job gets quantum
- - Quantum size is tricky choice
- - More overhead than FIFO

Assumptions: Quantum=2, Switch=1

Round Robin – Ready List

Job	t_a	CPU
P_0	0	8
P_1	2	4
P_2	4	3
P_3	6	5

Time	Ready List
0	P_0
2	
3	
5	
6	
8	

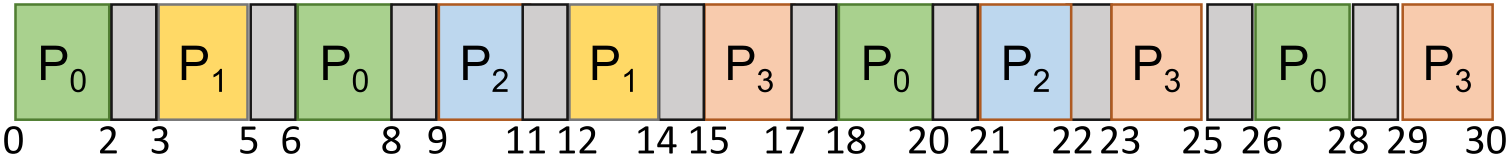
Round Robin

0	P_0	12	P_1	23	P_3
2	K	14	K	25	K
3	P_1	15	P_3	26	P_0
5	K	17	K	28	K
6	P_0	18	P_0	29	P_3
8	K	20	K	30	K
9	P_2	21	P_2		
11	K	22	K		

Round Robin

Job	t_a	CPU	t_e	t_d	Resp	T/A
P_0	0	8	0	28	0	28
P_1	2	4	3	14	1	12
P_2	4	3	9	22	5	18
P_3	6	5	15	30	9	24
				AVERAGE	3.75	20.50
				OVERHEAD	10	

t_a = arrival time
t_e = execution start time
t_d = departure time
Resp = response time
T/A = turnaround time



Shortest Job First (SJF)

Non-Preemptive

Schedule job with shortest CPU expected

- \+ Provably optimal (conditions)
- \+ Little overhead
- \- Poor response time
- \- Long jobs could starve

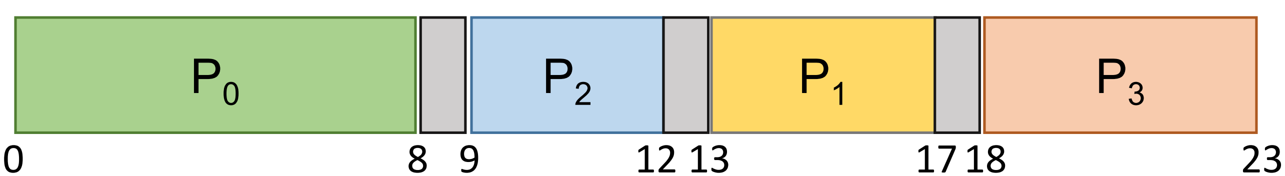
SJF

0	P_0
8	K
9	P_2
12	K
13	P_1
17	K
18	P_3
23	K

SJF

Job	t_a	CPU	t_e	t_d	Resp	T/A
P_0	0	8	0	8	0	8
P_1	2	4	13	17	11	15
P_2	4	3	9	12	5	8
P_3	6	5	18	23	12	17
				AVERAGE	7.00	12.00
				OVERHEAD	3	

t_a = arrival time
t_e = execution start time
t_d = departure time
Resp = response time
T/A = turnaround time



SRTCF

Shortest Remaining Time to Completion First

Preemptive version of SJF

Schedule job with shortest CPU expected

Preempt and reschedule when

- Job terminates
- Job arrives
- Internal event (yield)

Evaluation

+ Provably optimal AVG resp (conditions)
+ Short jobs preempt long jobs (Why?)
- Unfair
- Long jobs could starve

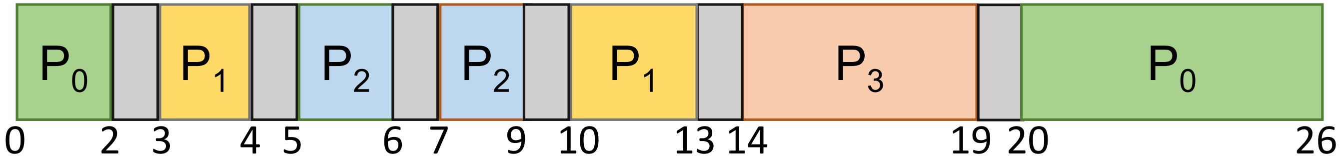
SRTCF

0	P_0
2	K
3	P_1
4	K
5	P_2
6	K
7	P_2
9	K
10	P_1
13	K
14	P_3
19	K
20	P_0
26	K

SRTCF

Job	t_a	CPU	t_e	t_d	Resp	T/A
P_0	0	8	0	26	0	26
P_1	2	4	3	13	1	11
P_2	4	3	5	9	1	5
P_3	6	5	14	19	8	13
				AVERAGE	2.50	13.75
				OVERHEAD	6	

t_a = arrival time
t_e = execution start time
t_d = departure time
Resp = response time
T/A = turnaround time



Problems

Where does kernel get CPU requirement?

- User?
  - user specifies time requirement
  - not realistic, because users may not know time requirement
- System?
  - previous executions (stats, history)
  - I/O in past → I/O in future (probably)
  - no help if random behavior

Adaptive Policies

Use past behavior to predict future performance

Changes to reflect current knowledge

Incorporates "*I/O before Compute*" idea

Two examples

- Multilevel feedback queuing
- Lottery scheduling

Multilevel Feedback Queuing

N queues/“levels” numbers 1 – N

Queue has priority and quantum

Quantum is “*exponentially increasing*”

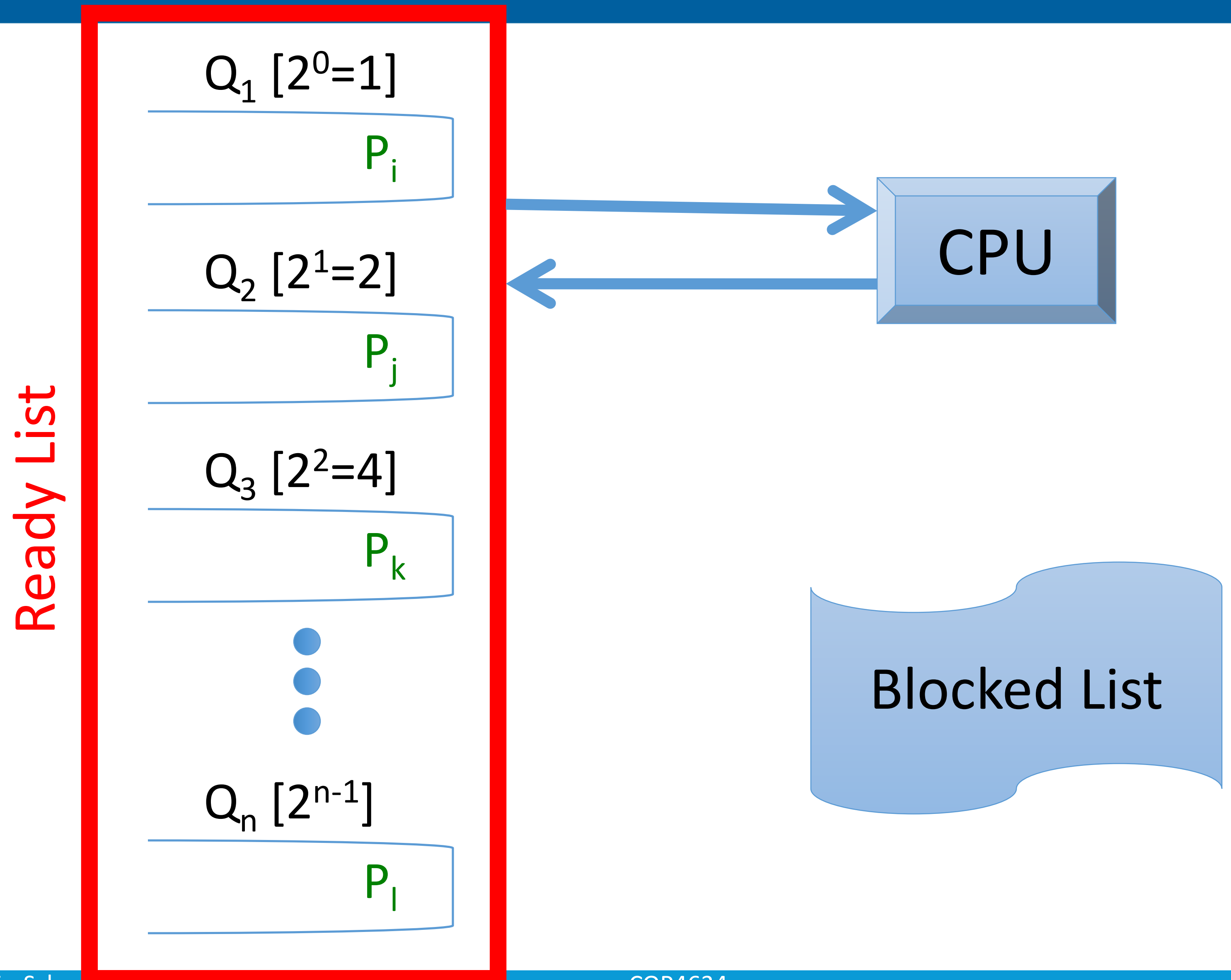
Q_i has priority *i* and quantum 2^{i-1}

Lower number is higher priority

All new processes start at priority 1

Priority altered based on process behavior

MLFQ Overview



MLFQ Algorithm

Scheduler picks process from highest priority non-empty queue

- If Q_i is empty, try Q_{i+1}

Process goes to CPU from Q_i

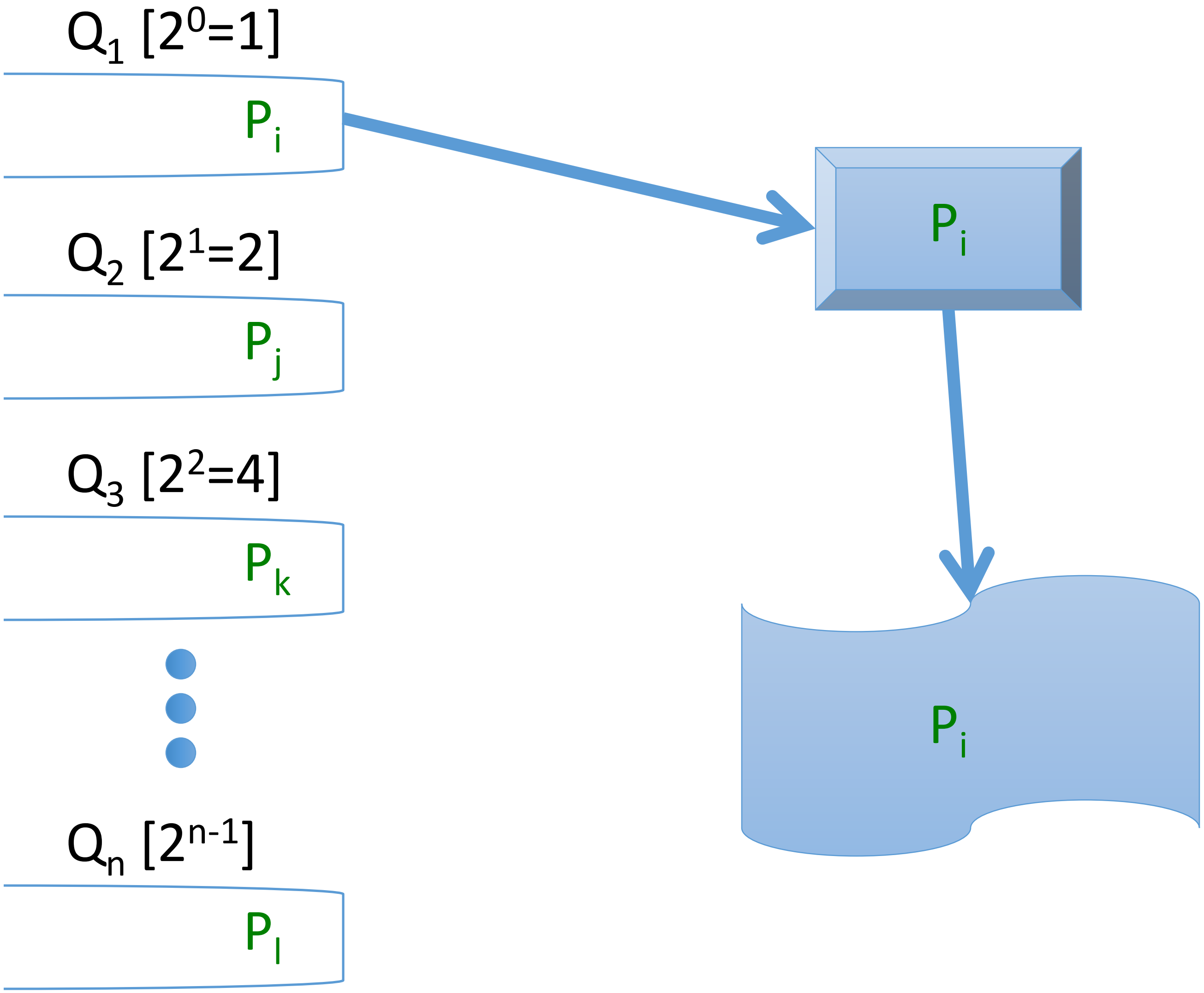
If quantum expires

- Add process to Q_{i+1}

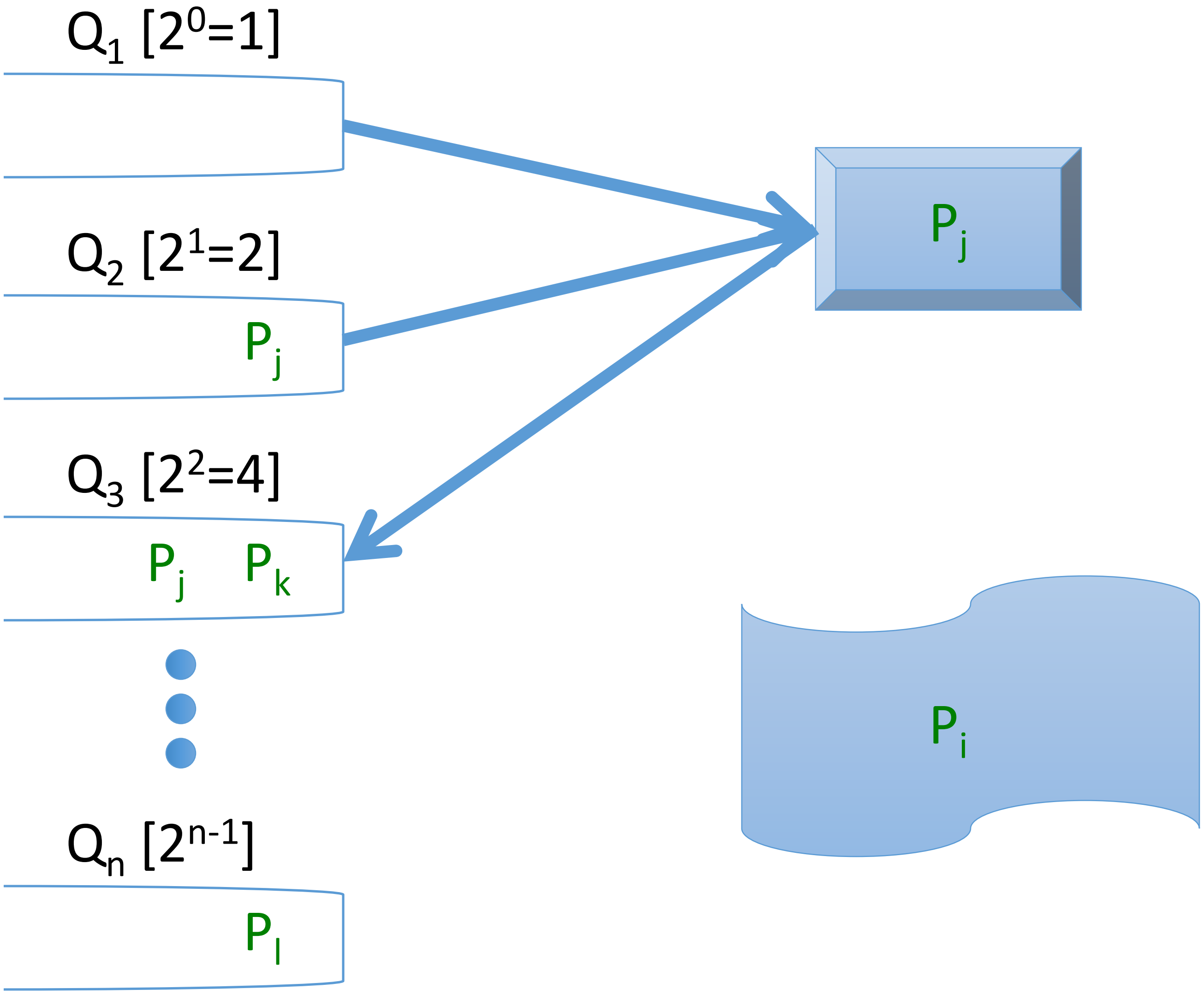
If I/O initiated

- Move process to blocked list
- Return process to either Q_{i-1} or Q_1

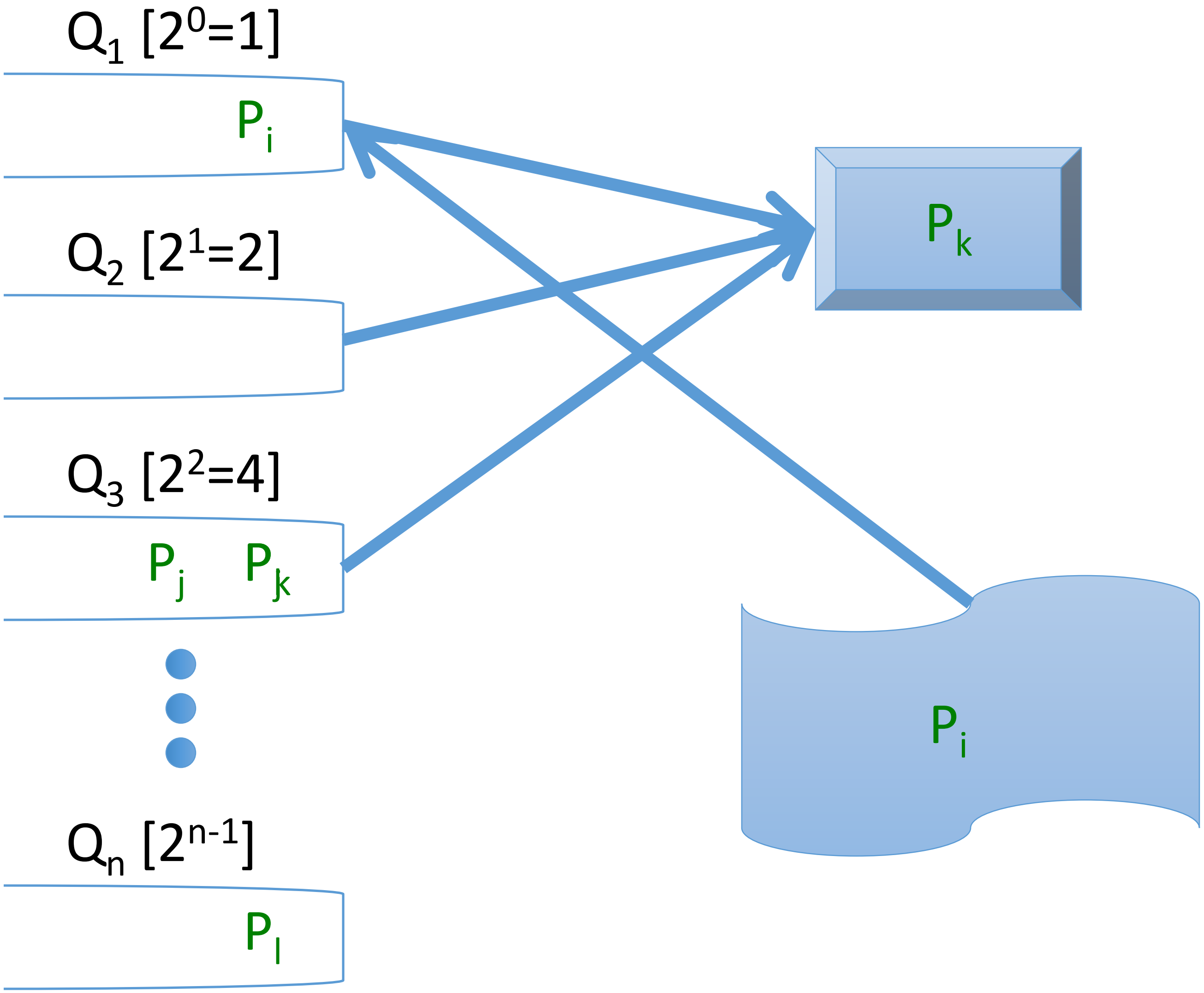
MLFQ - I/O Initiated



MLFQ - Compute



MLFQ – I/O Complete



Intended Effect

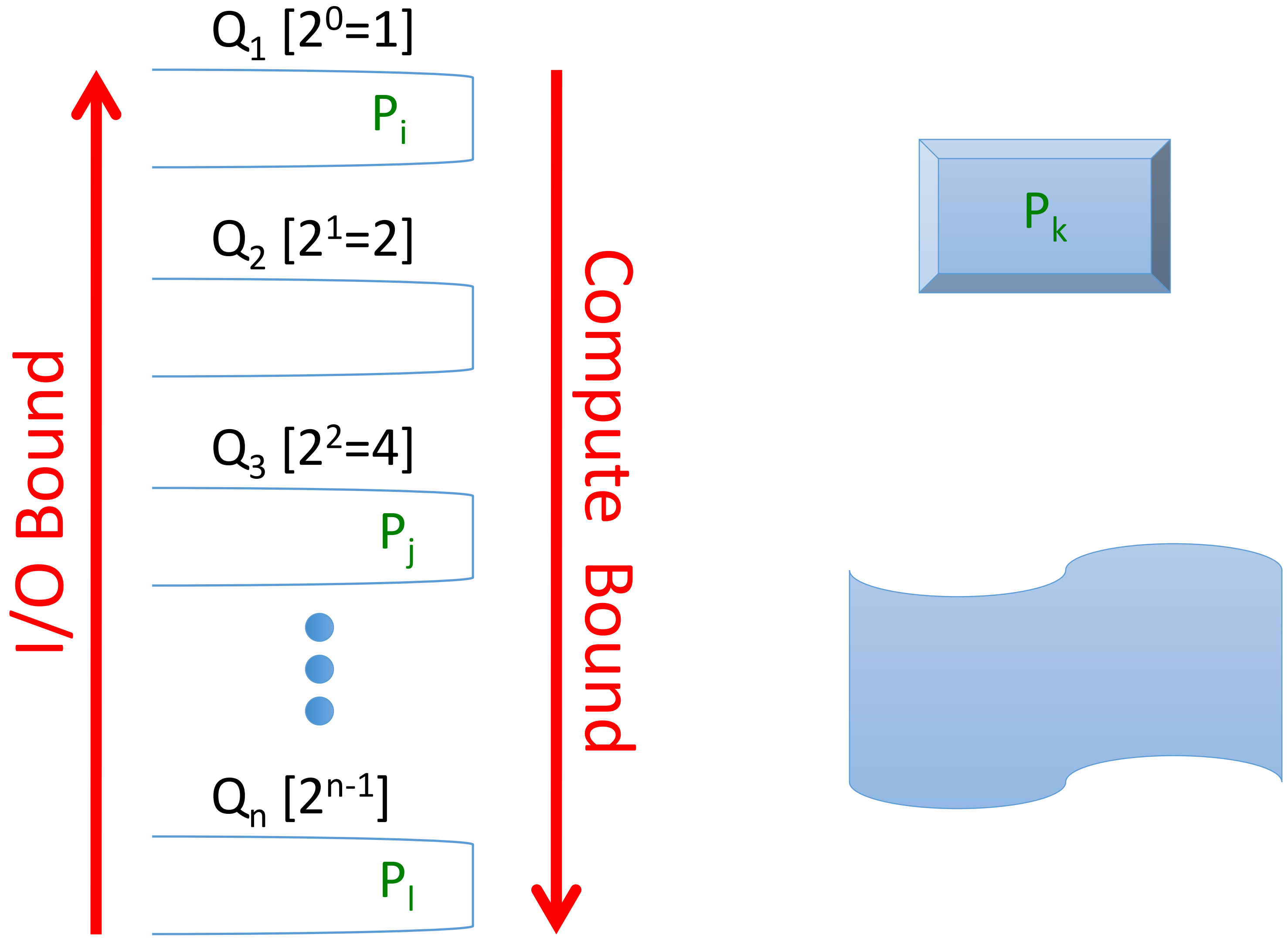
Compute-bound jobs

- Move to low priority
- Scheduled less frequently
- Get longer quanta

I/O-bound jobs

- Move to high priority
- Scheduled more frequently
- Get shorter quanta

MLFQ – Effect



Example

Assumptions:

- We have 4 queues
- 3 jobs arrive at time 0
- P_0: (2 CPU + 4 I/O) x 3
- P_1: (4 CPU + 2 I/O) x 3
- P_2: (20 CPU + 0 I/O) x 1

Lots of data to keep

Use an array w/ 4+Q columns

MLFQ

T	Q_1 1	Q_2 2	Q_3 4	Q_4 8	CPU	Wait	
0	P_1 P_2				P_0		All t_a
1	P_2	P_0			P_1		
2		P_0 P_1			P_2		
3		P_1 P_2			P_0		
4		P_2			P_1	$P_0$8	
6			P_1		P_2	$P_0$8	
8	*P_0*		P_1 P_2		P_0		
9		*P_0*	P_1 P_2		P_0		
10			P_2		P_1	$P_0$14	

MLFQ

T	Q_1 1	Q_2 2	Q_3 4	Q_4 8	CPU	Wait	
10			P_2		P_1	$P_0$14	
11					P_2	$P_0$14 $P_1$13	
13		P_1			P_2	$P_0$14	
14	P_0	P_1			P_2		
15		P_1		P_2	P_0		
16		P_0		P_2	P_1		
18			P_1	P_2	P_0		
19				P_2	P_1	$P_0$23	

MLFQ

T	Q_1 1	Q_2 2	Q_3 4	Q_4 8	CPU	Wait	
19				P_2	P_1	$P_0$23	
21					P_2	$P_0$23 $P_1$23	
23		P_1			P_2		P_0t_d
29				P_2	P_1		
31			P_1	P_2	P_1		
33					P_2	$P_1$35	
35					P_2		P_1t_d
38							P_2t_d

Multilevel Feedback Queuing

Job	t_a	CPU	t_e	t_d	Resp	T/A
P_0	0	6	0	23	0	23
P_1	0	12	1	35	1	35
P_2	0	20	2	38	2	38
				AVERAGE	1.00	32.00

t_a = arrival time
t_e = execution start time
t_d = departure time
Resp = response time
T/A = turnaround time

Implementation Notes

Countermeasure: meaningless I/O

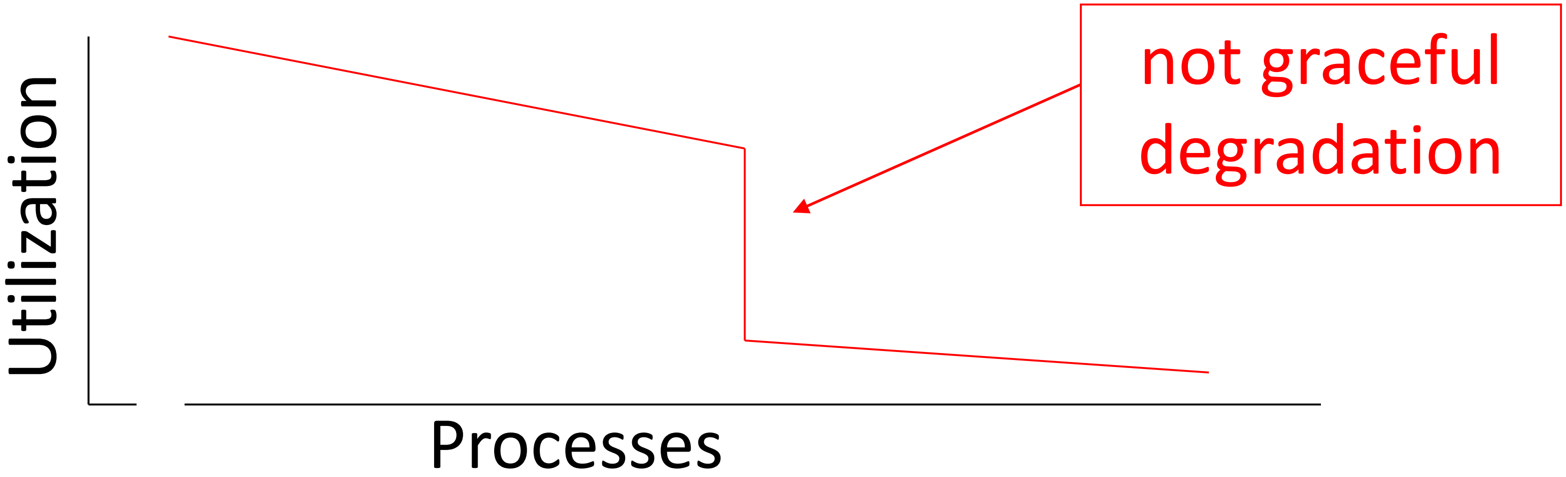
Problem: Enough I/O & compute jobs starve

Unix "*fix*" (process aging)

- Set *time* when process added to Q_i
- If *time* expires before service provided
  - Move process to Q_{i-1}
  - Reset timer

Implementation Notes

- Overloaded system
  - All jobs move to Q1
  - Interactive jobs stop responding
  - Compute jobs get very little done
  - System utilization suffers



Lottery Scheduling

All processes initially assigned x "*tickets*"

Scheduler randomly picks a winning ticket

Process with winning ticket is scheduled

Quantum Expires

- Some tickets taken away from process

I/O initiated

- Some tickets given to process

Advantages to Lottery

Average CPU time for process is proportional to tickets in system

Adding jobs effects all other jobs proportionately

All processes have at least 1 ticket

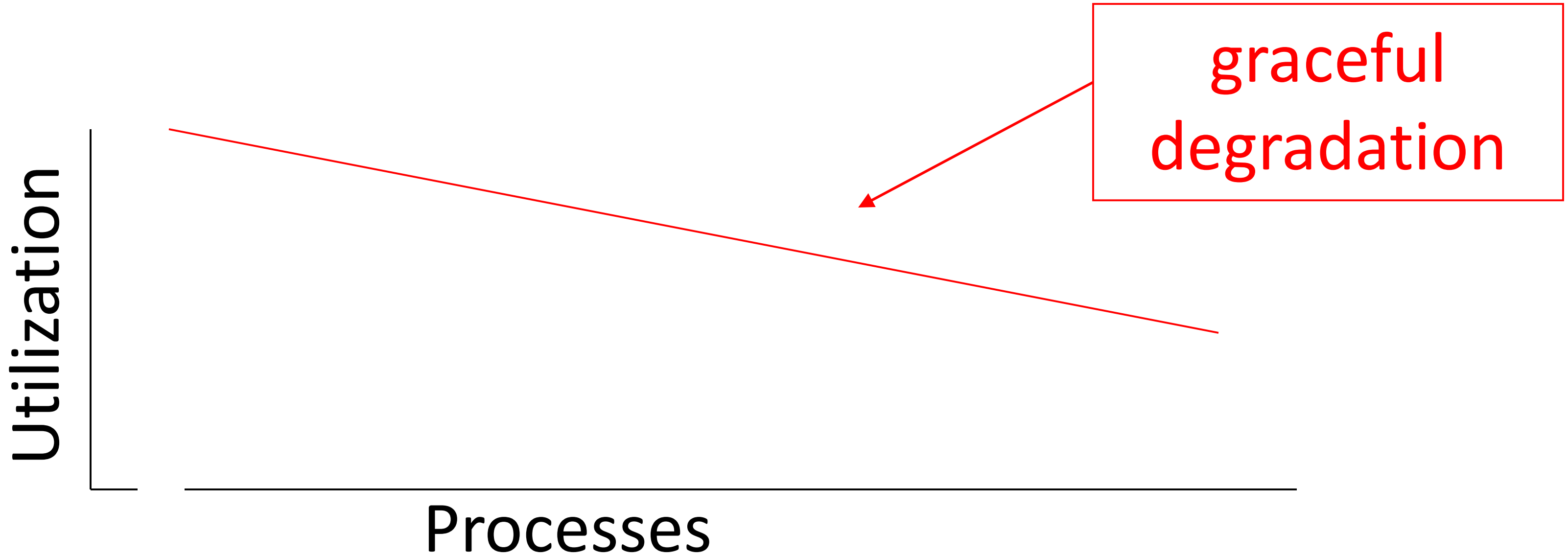
- Statistically, no starvation
- Never remove last ticket from process
- Never give more than Max tickets

Graceful degradation

Degradation of Utilization

Examples (L (long) = 1 ticket, S (short) = 10 tickets)

- 1L & 1S: L (1/11), S (10/11)
- 1L & 2S: L (1/21), S (10/21)
- 1L & 5S: L (1/51), S (10/51)
- 1L & 10S: L (1/101), S(10/101)



Summary

- Scheduler selects processes for execution.
  - must keep all systems equally busy
  - must give every process a chance to run
- Metrics to measure performance of algorithms include:
  - response time
  - turn-around time
  - overhead
- Best scheduling algorithms strive to optimize run-time performance for mix of IO- and CPU-bound processes.