

Module 4 – CPU Scheduling

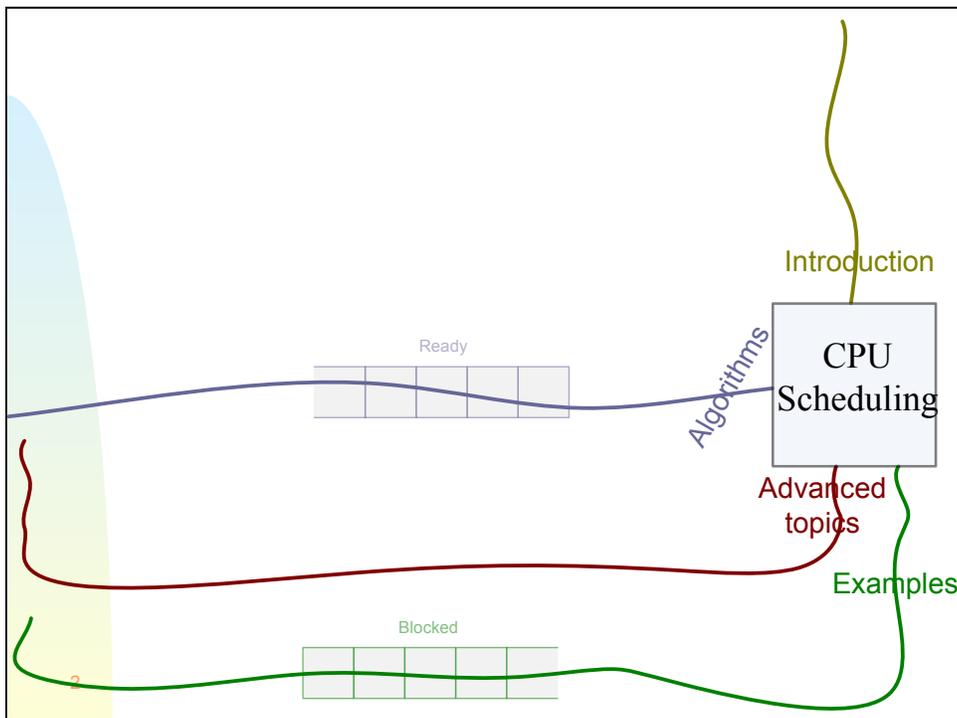
Reading: Chapter 5

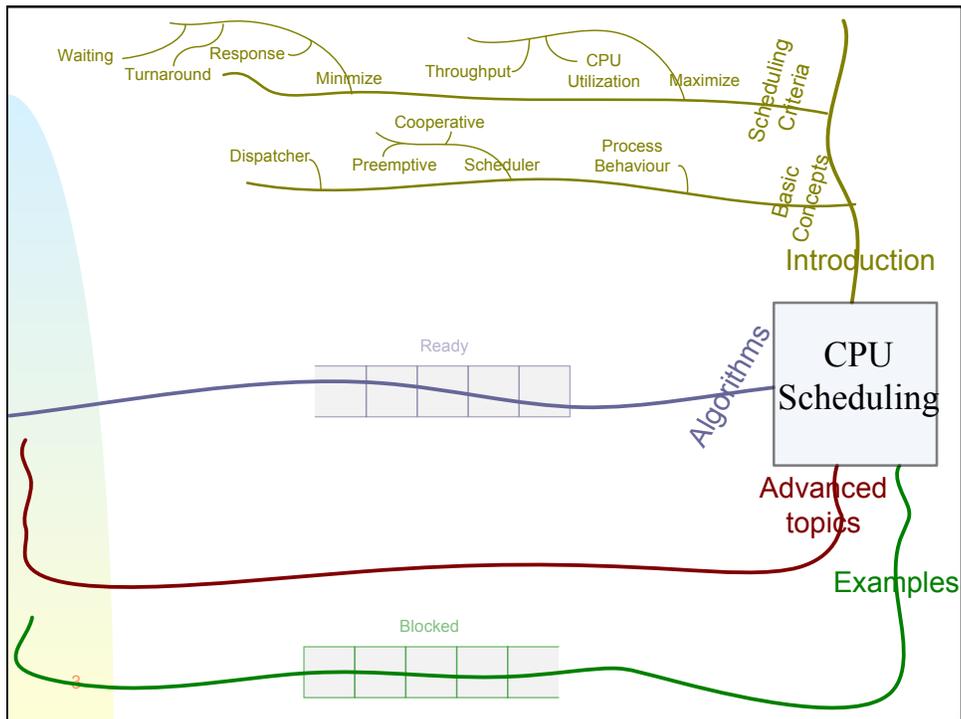
Providing some motivation and scheduling criteria.

Shall study a number of different algorithms used by CPU (short-term) schedulers.

Overview some advance topics such as multiprocessor scheduling and evaluating algorithms.

Examine how scheduling is done in some real systems.





Basic Concepts

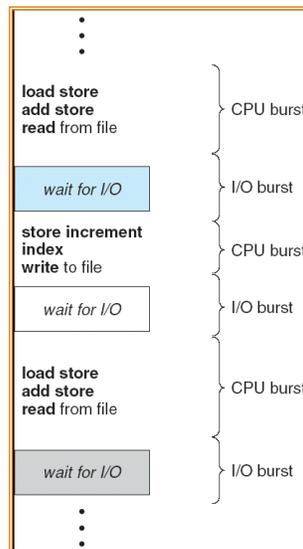
Why do we want multiprogramming?

- To efficiently use the computer resources (CPU, I/O devices)
- I.e. when one process blocks, assign the CPU to another process
- That means we need to decide which process to execute, and then give the control to it
- If we want efficient resource utilization, we need CPU scheduling
 - CPU scheduling principles can be applied to scheduling of other resources (I/O devices, etc.)

Process behaviour

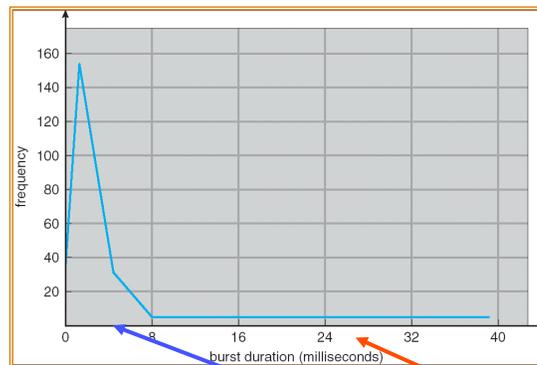
- Different processes might have different needs of resources
- Process execution can be seen as alternating bursts of CPU execution and I/O wait
- If we want intelligent scheduling decisions, we need to understand process behaviour

Closer look at process behaviour



5

Histogram of CPU-burst Times



- **Experimental observation:**
 - In a typical system, we normally see a large number of short CPU bursts, and a small number of long CPU bursts
- CPU bound processes are normally composed of a few long CPU bursts.
- I/O bound processes are normally composed of many short CPU bursts

6

CPU (Short term) Scheduler

- Selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them
- When may (should) the CPU scheduling decisions take place?
 1. A process switches from running to waiting state
 - made an I/O request
 3. A process terminates
 4. A process switches from running to ready state
 - used up its allotted time slice
 5. A process switches from waiting to ready
 - I/O operation completed
- Preemptive scheduling
 - The CPU is taken from the currently running process before the process voluntarily relinquishes it
 - Which of the above cases involve preemptive scheduling?

7

Dispatcher

- Dispatcher module gives control of the CPU to the process selected by the short-term scheduler; this involves:
 - switching context
 - switching to user mode
 - jumping to the proper location in the user program to restart that program
- **Dispatch latency** – time it takes for the dispatcher to stop one process and start another running
 - we want this to be really low

8

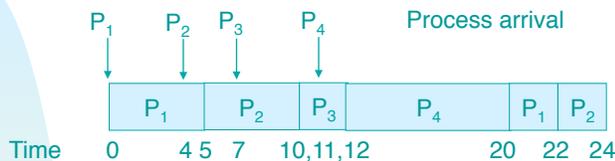
Scheduling Criteria

What kind of criteria we might want the scheduler to optimize?

- Main reason for multiprogramming?
 - resource utilization – keep the CPU and I/O as busy as possible
- On time-shared systems?
 - Response time – amount of time it takes from when a request was submitted until the first response is produced (starts to output)
- On servers?
 - Throughput – # of processes that complete their execution per time unit
- On batch systems?
 - **Batch processing is execution of a series of programs ("jobs") on a computer without manual intervention.**
 - Turnaround time – amount of time from job submission to its completion
- On heavily loaded systems
 - Fairness/Waiting time – amount of time a process has been waiting in the ready queue
- Averages are often considered

9

Scheduling Criteria Example



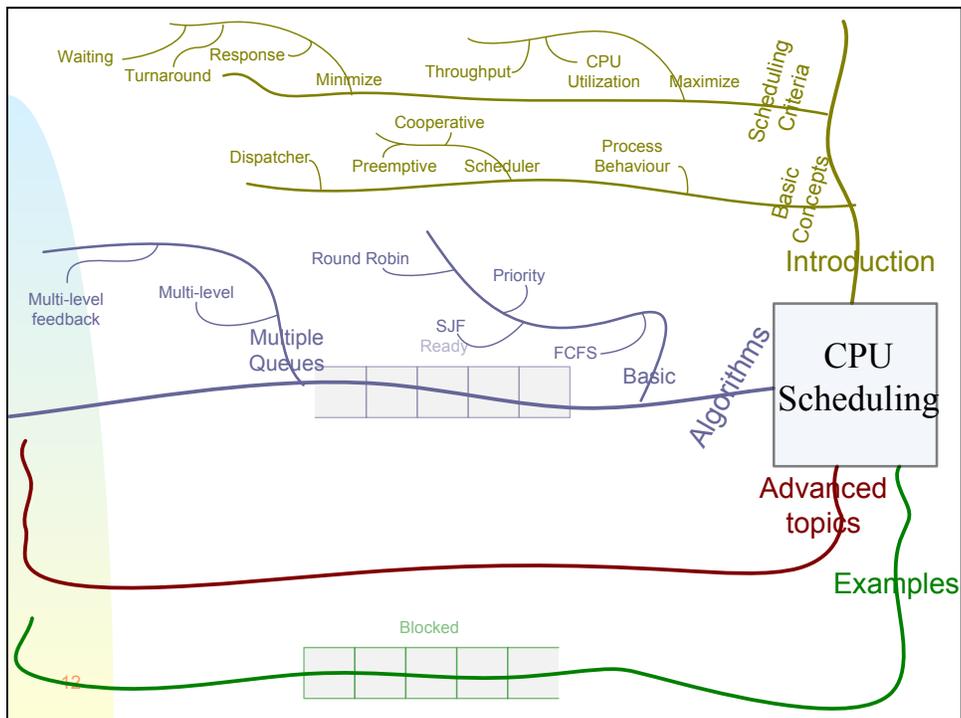
- **CPU utilization:**
 - 100%
- **Throughput :**
 - 4/24
- **Turnaround time (P₃, P₂):**
 - P₃: 5
 - P₂: 20
- **Waiting time (P₂):**
 - P₂: 13 = 1 (5-4) + 12 (22-10)
- **Response time (P₃, P₂):**
 - P₃: 3
 - P₂: 1

10

Optimization Criteria

- **Maximize**
 - CPU utilization
 - throughput
- **Minimize**
 - turnaround time
 - waiting time
 - response time

11



12

Now, let's see some algorithms!

What is the simplest scheduling algorithm you can imagine?

- First-Come, First-Serve

How does FCFS work?

- When a process arrives, it is put at the end of the ready queue
- The running process runs until its CPU burst is finished
- This process is moved out of CPU into I/O queue
- Next job gets CPU for its CPU burst

Is it preemptive?

- No

Let's see an example:

13

First-Come, First-Served (FCFS) Scheduling

<u>Process</u>	<u>Burst Length</u>	<u>Arrival Time</u>
P_1	24	0 (first)
P_2	3	0 (second)
P_3	3	0 (third)



Waiting time for $P_1 = 0$; $P_2 = 24$; $P_3 = 27$
Average waiting time: $(0 + 24 + 27)/3 = 17$

14

FCFS Scheduling (Cont.)

Suppose that the processes arrive in the order

P_2, P_3, P_1

The schedule now looks this way:



Waiting time for $P_1 = 6$; $P_2 = 0$; $P_3 = 3$
Average waiting time: $(6 + 0 + 3)/3 = 3$
Much better than the previous case

15

Convoy Effect with the FCFS

- Consider a single CPU bound process and many I/O bound processes (fairly normal situation).
- **Many I/O finish while waiting CPU bound process to finish**
- **finished I/O bound processes become ready and then wait for the CPU:**
- I/O under utilized (*).
- The CPU bound process requests I/O: the other processes rapidly complete CPU bursts and go back to I/O: CPU under utilized.
- CPU bound process finishes I/O, and so do the other processes: back to *
- Solutions?
- **One process consists of CPU and I/O bursts**

FCFS Scheduling Discussion

Is it simple and easy to program?

- Yes!

Can we find other benefits?

- It costs little time to make scheduling decision

Does it provide low waiting time?

- Not at all, can be quite awful

Does it provide good CPU utilization?

- No – the convoy effect

OK, forget about it.

Now, let's try to reduce the average waiting time

Idea: It is better to execute ready processes with short CPU bursts

17

Shortest-Job-First (SJR) Scheduling

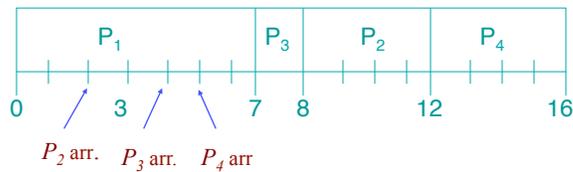
- Associate with each process the length of its next CPU burst. Use these lengths to schedule the process with the shortest time
- Two schemes:
 - nonpreemptive – once CPU given to the process it cannot be preempted until completes its CPU burst
 - preemptive – if a new process arrives with CPU burst length less than remaining time of current executing process, preempt. This scheme is also known as the Shortest-Remaining-Time-First (SRTF)
 - *Just call it preemptive SJF*
- Preemptive SJF is optimal – gives minimum average waiting time for a given set of processes
 - Moving a process with short CPU burst in front of a process with longer CPU burst reduces average waiting time
 - **Is it always correct? Counter-example?**

18

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

- SJF (non-preemptive)



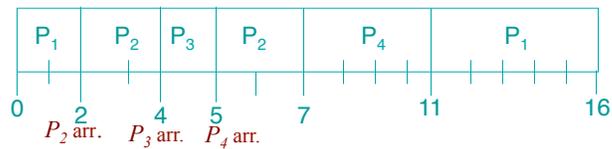
- Average waiting time = $(0 + 6 + 3 + 7)/4 = 4$

19

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

- SJF (preemptive)



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

20

Small technical detail with SJF

How do we know the length of the next CPU burst?

- If you know that, you probably know also tomorrow's stock market prices...
 - You will be rich and won't need to waste time in CSI3131 class
- So, we can only estimate it

Any idea how to estimate the length of the next CPU burst?

- Probably similar as the previous bursts from this process
- Makes sense to give more weight to the more recent bursts, not just straightforward averaging
 - Use exponential averaging

21

Exponential averaging with SJF

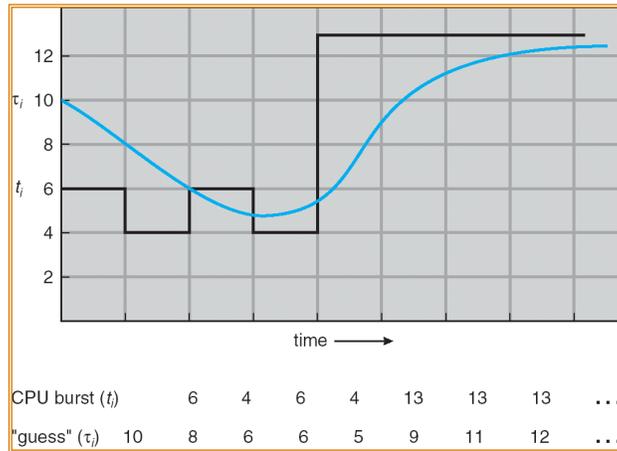
1. t_n = actual length of n^{th} CPU burst
2. τ_{n+1} = predicted value for the next CPU burst
3. α , $0 \leq \alpha \leq 1$ = relative weight of recent vs past history
4. Define:

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

$$\tau_{n+1} = \alpha t_n + (1 - \alpha)\alpha t_{n-1} + (1 - \alpha)^2 \alpha t_{n-2} + \dots \\ + (1 - \alpha)^i \alpha t_{n-i} + \dots + (1 - \alpha)^n t_1$$

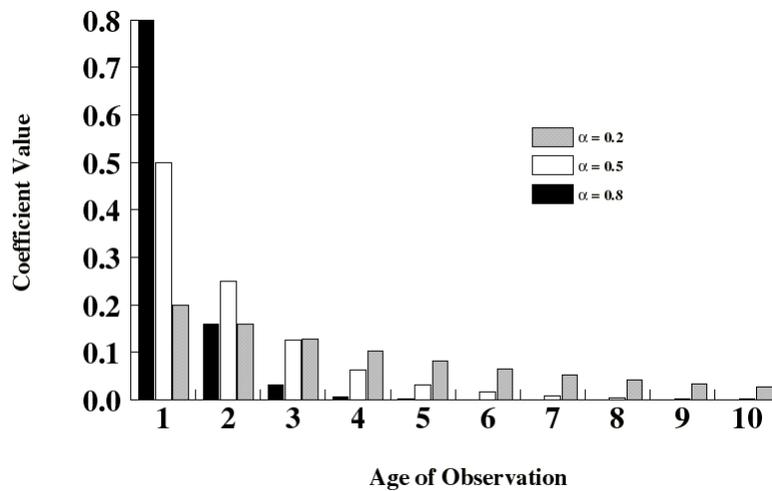
22

Prediction of the Length of the Next CPU Burst



23

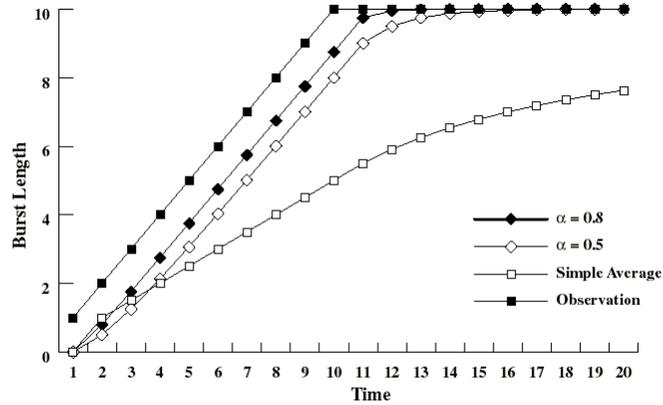
Exponential decrease of coefficients [Stallings]



24

Stallings

Exponential decrease of coefficients [Stallings]

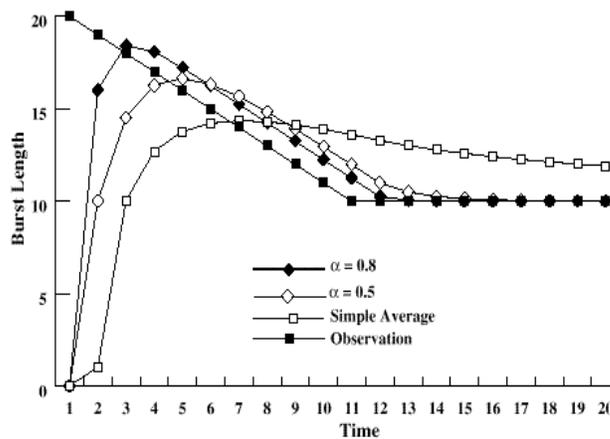


- $t_0 = 0$ (processes may be categorized first for separate e.a.)
- A larger coefficient leads to estimates that react more rapidly to changes in the process behaviour

25

Stallings

A second example [Stallings]



(b) Decreasing function

26

Stallings

How to choose the coefficient α

- A small coefficient is advantageous when a process can have anomalies in behaviour, after which it returns to previous behaviour (must ignore recent behaviour).
 - Limit case: $\alpha = 0$, use only the initial estimate
- A large coefficient is advantageous when a process is susceptible to changing rapidly from one type of activity to another.
 - Limit case: $\alpha = 1$, $\tau_{n+1} = t_n$
 - The last burst is the only one used for estimating the next one.

27 ▪ So how to choose it in practice?

SJF Discussion

Does it ensure low average waiting time?

- Yes, it was designed that way
 - As long as our burst-length predictions more-or-less work

Does it provide low response time?

- Not necessarily, if there is a steady stream of short CPU bursts, the longer bursts will not be scheduled
- This is called starvation
 - A process is blocked forever, always overtaken by other processes (well, or at least while the system is busy)

Let's see Priority Scheduling.

28

Priority Scheduling

- A priority number (usually integer) is associated with each process
 - On some systems (Windows), the higher number has higher priority
 - On others (Unix) , the smaller number has higher priority
- The CPU is allocated to the process with the highest priority
 - Can be preemptive or non-preemptive
 - but usually you want to preempt low-priority process when a high priority process becomes ready
- Priority can be explicit
 - Assigned by the sysadmin/programmer, often for political reasons
 - Professor jobs are of higher priority
 - But also for technical reasons
 - This device has to be serviced really fast, otherwise the vital data will be lost (real-time processing)
- Priority can also be implicit (computed by the OS)
 - SJF can be seen as priority scheduling where priority is the predicted next CPU burst time

29

Priority Scheduling Discussion

Good properties

- Professor jobs will be scheduled before student jobs
- Allows support of real-time processing

Bad properties

- Professor jobs will be scheduled before student jobs
 - OK, give me something else
- starvation – low priority processes may never execute

How to resolve the starvation problem?

- aging – keep increasing the priority of a process that has not been scheduled for a long time

What to do with the processes of the same priority level?

- FCFS
- Might as well add preemption = Round Robin

30

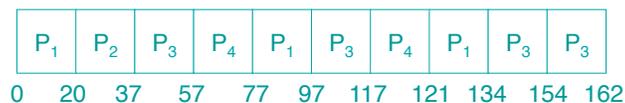
Round Robin (RR)

- Each process gets a small unit of CPU time (*time quantum, time slice*), usually 10-100 milliseconds.
 - After this time has elapsed, the process is preempted and added to the end of the ready queue.
 - The CPU is then given to the process at the head of the 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 FCFS
 - q small \Rightarrow too much context switching overhead
 - q must be large with respect to context switch time

31

Example of RR with Time Quantum = 20

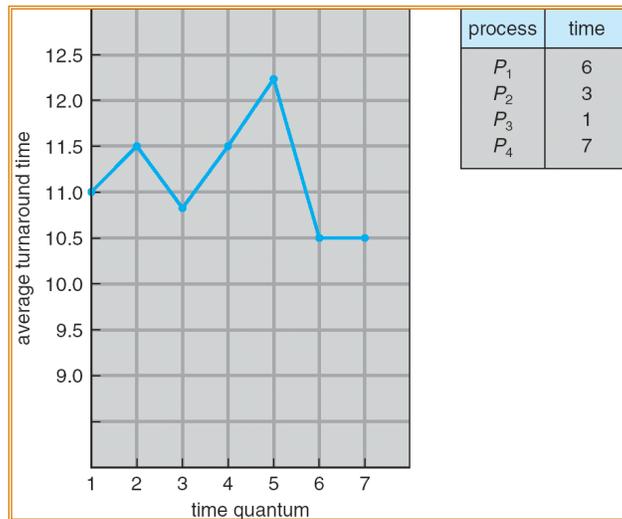
<u>Process</u>	<u>Burst Time</u>
P_1	53
P_2	17
P_3	68
P_4	24



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

32

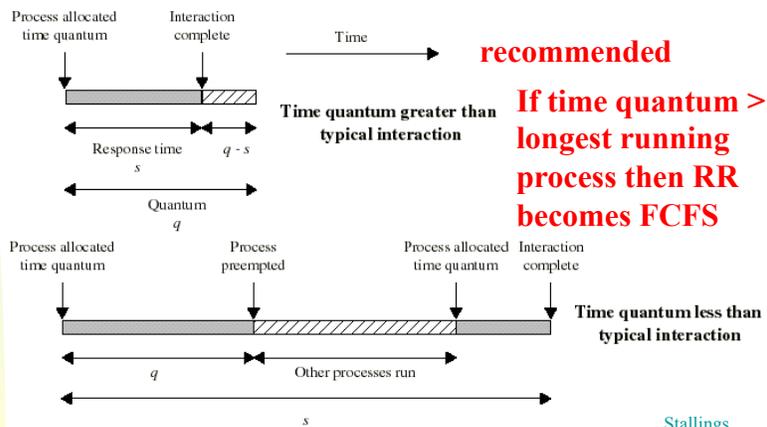
Turnaround Time Varies With The Time Quantum



33

Selecting quantum for Round Robin [Stallings]

- Must be much larger than time for executing context switching
- Must be larger than the typical CPU burst length (to give time for most processes to complete their burst, but not so long to penalize the processes with short bursts).



34

Algorithms we have seen so far

- **First Come First Serve**
 - simple, little overhead, but poor properties
- **Shortest Job First**
 - needs to know CPU burst times
 - exponential averaging of the past
- **Priority Scheduling**
 - This is actually a class of algorithms
- **Round Robin**
 - FCFS with preemption

35

Let's see how they work – Tutorial Exercise

Consider three processes P1, P2, P3

- Burst times for P1: 14,12,17
- Burst times for P2: 2,2,2,3,2,2,2,3,2,2,2,3,2,2,2,3
- Burst times for P3: 6,3,8,2,1,3,4,1,2,9,7
- All three arrive at time 0, in order P1, P2, P3
- Each CPU burst is followed by an I/O operation taking 6 time units
- Let's simulate the scheduling algorithms
 - FCFS
 - Round Robin (quantum=5)
 - Non-preemptive SJF or Preemptive SJF (your choice)
 - Round robin (quantum=5) with Priority scheduling, priorities are $P2=P3>P1$

36

Multilevel Queue

Idea: **Partition the ready queue into several queues, and handle each queue separately.**

Example:

- foreground queue (interactive processes)
- background (batch processes)

Each queue might have its own scheduling algorithm

- foreground – RR (for low response time)
- background – FCFS (for simplicity and low context-switch overhead)

37

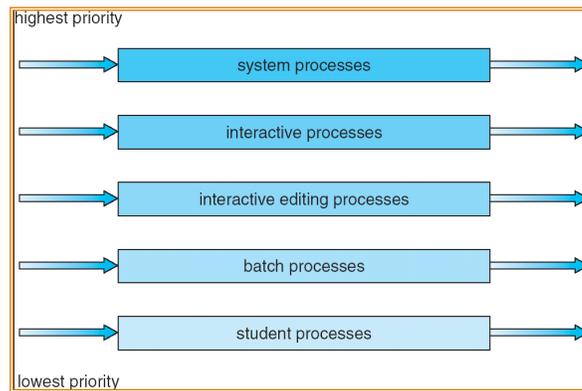
Multilevel Queue

How to schedule from among the queues?

- **Fixed priority scheduling**
 - i.e. the processes from foreground queue get the CPU, the background processes get the CPU only if the foreground queue is empty
 - Possibility of starvation.
- **Time slice – each queue gets a certain amount of CPU time which it can schedule amongst its processes**
 - i.e., 80% to foreground queue, 20% to background queue
 - not necessarily optimal

38

Multilevel Queue Scheduling



39

Multilevel Feedback Queue

Idea:

- **Use multilevel queues**
- **A process can move up and down queue hierarchy**
 - **Why would a process move to a lower priority queue?**
 - It is using too much CPU time
 - **Why would a process move to a higher priority queue?**
 - Has been starved of CPU for long time
 - A way to implement aging

40

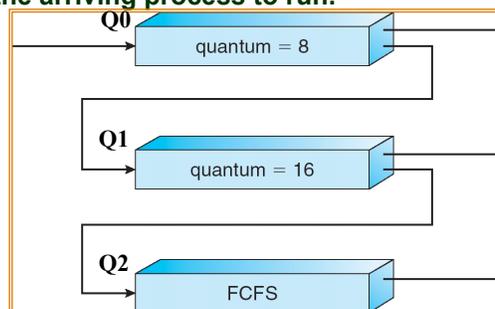
Multilevel Feedback Queue

- **Multilevel-feedback-queue scheduler defined by the following parameters:**
 - number of queues
 - scheduling algorithms for each queue
 - method used to determine when to upgrade a process
 - method used to determine when to demote a process
 - method used to determine which queue a process will enter when that process needs service
- **This algorithm is the most general one**
 - It can be adapted to specific systems
 - But it is also the most complex algorithm

41

Example of Multilevel Feedback Queue

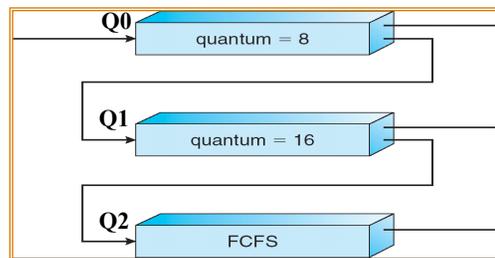
- **Scheduler selects processes in Q0 first (highest priority)**
 - If Q0 is empty, the processes from Q1 are selected.
 - If both Q0 and Q1 are empty, processes from Q2 are selected
- **If a process arrives in a higher priority queue when another from a lower priority queue is running, the running process will be preempted, to allow the arriving process to run.**
- **When a process exhausts its quantum in either Q0 or Q1, it is preempted and moved to the lower priority queue.**



42

Example of Multilevel Feedback Queue

- **Scheduling example**
 - A new job enters queue Q_0 which is served FCFS. When it gains CPU, job receives 8 milliseconds.
 - If it does not finish in 8 milliseconds, job is preempted and moved to queue Q_1 and served again FCFS to receive another 16 additional milliseconds.
 - If it still does not complete, it is preempted and moved to queue Q_2 .



43

Multilevel Feedback Queue Discussion

Exact properties depend on the parameters

Flexible enough to accommodate most requirements

The convoy example:

- One process with long CPU burst time
- Several I/O bound processes with short CPU burst time
- Even if the all processes start at the same level, the CPU-intensive process will be soon demoted to a low priority queue
- The I/O bound processes will remain at high priority and will be swiftly serviced, keeping the I/O devices busy

44

Let's simulate the multi-level feedback queue – Tutorial Exercise

Consider three processes P1, P2, P3

- Burst times for P1: 14,12,17
- Burst times for P2: 2,2,2,3,2,2,2,3,2,2,2,3,2,2,2,3
- Burst times for P3: 6,3,8,2,1,3,4,1,2,9,7
- All three arrive at time 0, in order P1, P2, P3
- Each CPU burst is followed by an I/O operation taking 6 time units
- Parameters:
 - Queue 0 – quantum of 2 time units
 - Queue 1 – quantum of 4 time units
 - Queue 2 – FCFS

45

The scheduling algorithms we have seen

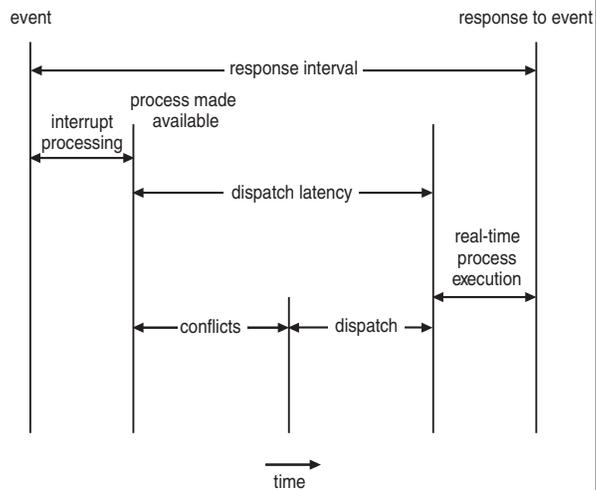
- **First Come First Serve**
 - simple, little overhead, but poor properties
- **Shortest Job First**
 - needs to know CPU burst times
 - exponential averaging of the past
- **Priority Scheduling**
 - This is actually a class of algorithms
- **Round Robin**
 - FCFS with preemption
- **Multilevel Queues**
 - Different scheduling algorithms possible in each queue
- **Multilevel Feedback Queues**
 - combines many ideas and techniques

46

Real-Time CPU Scheduling (Cont.)

- Conflict phase of dispatch latency:

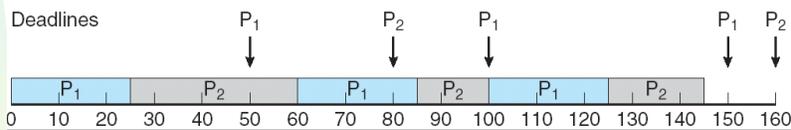
1. Preemption of any process running in kernel mode
2. Release by low-priority process of resources needed by high-priority processes

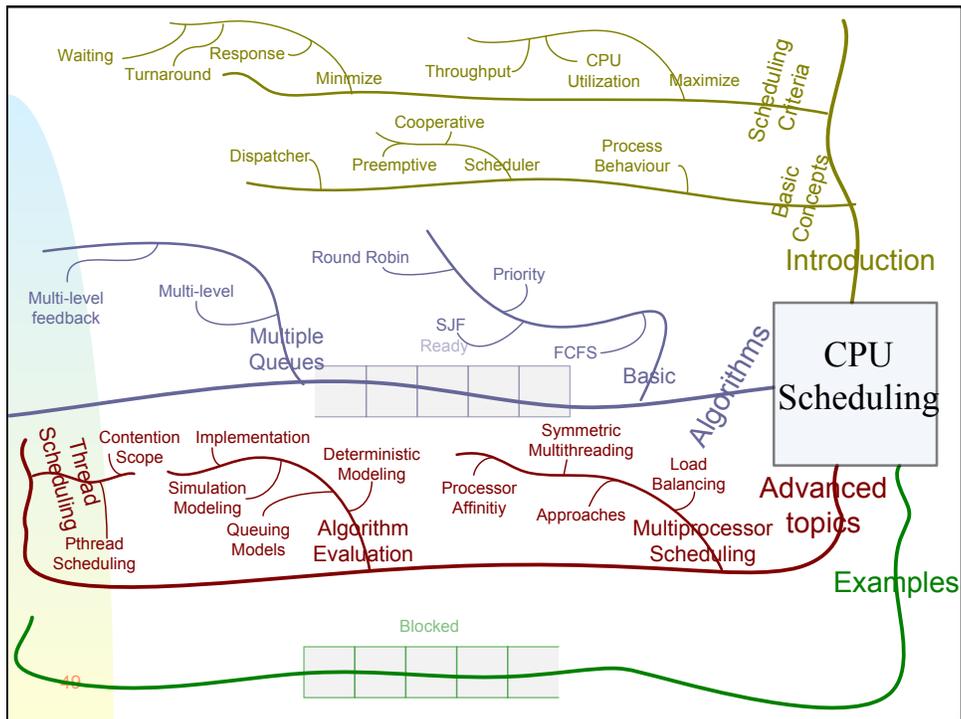


Earliest Deadline First Scheduling (EDF)

- Priorities are assigned according to deadlines:

the earlier the deadline, the higher the priority;





Multiple-Processor Scheduling

Good news:

- **With multiple CPUs, we can share the load**
- **We can also share the OS overhead**

Bad news:

- **We are expected to efficiently share the load**
- **Managing the OS structures gets more complicated**
- **The scheduling gets more complex**

Multiple-Processor Scheduling - Approaches

We assume *homogeneous processors*

- i.e. all processors have the same functionality

Still, the scheduler (and possibly the whole OS) might be run on a single CPU

- **Asymmetric multiprocessing** – only one CPU accesses the system data structures and makes scheduling decisions
 - alleviates the need for data sharing
 - but might become bottleneck
- **Symmetric multiprocessing (SMP)** – each CPU is self-scheduling
 - Either from a common ready queue, or from a private ready queue
 - Care must be taken when CPUs access common data structures
 - Virtually all modern operating systems support SMP including Windows XP, Solaris, Linux, Mac OS X.

51

SMP Scheduling Issues

- **Processor Affinity**
 - When process runs on a physical CPU, cache memory is updated with content of the process
 - If the process is moved to another CPU, benefits of caching is lost.
 - SMP systems try to keep processes running on the same physical CPU – know as processor affinity
- **Load Balancing**
 - Required when each CPU has its own Ready Queue
 - Push migration: a task is run periodically to redistribute load among CPUs (their ready queue)
 - Pull migration: An idle CPU (i.e. with an empty ready queue), will pull processes from other CPUs
 - Linux supports both techniques
 - Can counteract the benefits of processor affinity, since it moves processes from one CPU to another

52

Multi-Core and Multithreading

- Feature provided in hardware
- A single physical processor contains several logical CPUs
- **A single chip has multiple processor cores**
 - Achieved by replicating some resources such as the program counter (instruction pointer)
 - Big advantage is that many resources can be shared (such as arithmetic and logic units). ??
 - **But each core has its register set (separate logical CPU)**
- Each logical processor can be assigned different threads (**one or more**) to execute
- **One thread Compute, other thread memory access**
- To benefit from such hardware
 - Applications should be **SMP** aware
 - OS' s should differentiate between logical and physical processors e.g. schedule on different physical CPUs first
- Dual-core processors are examples of such processors

Thread Scheduling

Contention Scope

- **Process contention scope (PCS)**
 - The threads within a process compete for the LWPs (**lightweight process**), i.e. to attain the running state
 - **LWP: light-weight process for mapping user-level thread to kernel-level thread**
 - Managed by the threads library when many-to-one or many-to-many (**user to kernel thread**) model is used
- **system contention scope (SCS)**
 - All kernel threads compete for the CPU
 - Managed by the kernel – kernel decides which thread to schedule
 - Implied when one-to-one model is used

54

PThread Scheduling

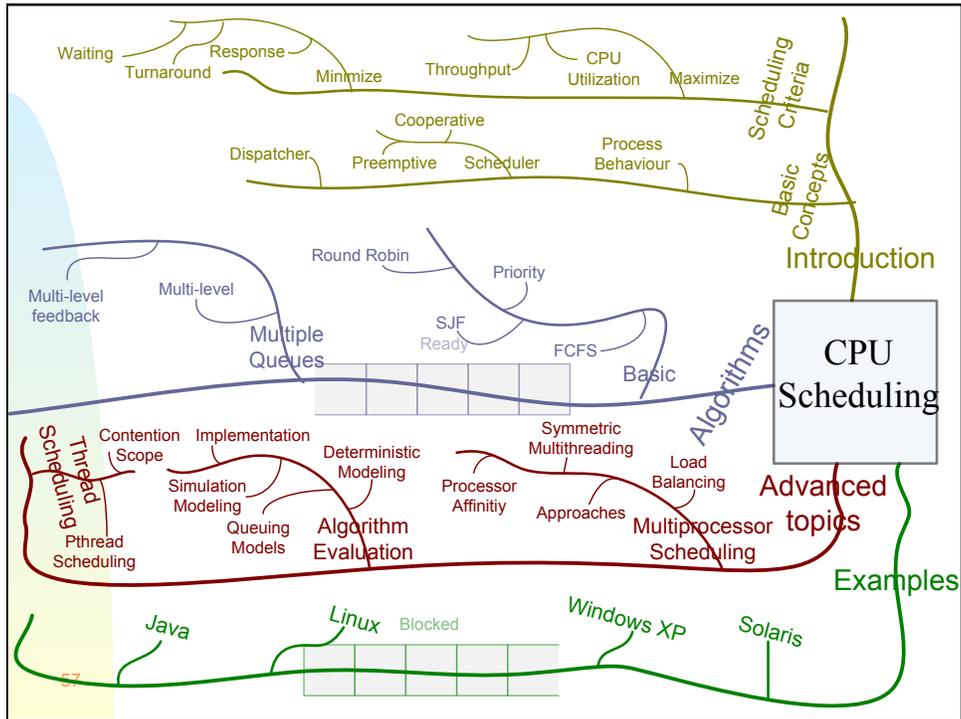
- The POSIX Pthread standard provides the means to specify either PCS or SCS scopes
 - See next slide for coding example.
- On systems that support a many-to-many threading model
 - PTHREAD_SCOPE_PROCESS policy schedules user-level threads onto available LWPs
 - PTHREAD_SCOPE_SYSTEM policy creates and binds an LWP to each user-level thread
- On some systems, only one scope is supported
 - E.g. Linux and Mac OS X only support PTHREAD_SCOPE_SYSTEM.

55

Pthread Scheduling API

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5
int main(int argc, char *argv[])
{
    int i;
    pthread_t tid[NUM_THREADS];
    pthread_attr_t attr;
    /* get the default attributes */
    pthread_attr_init(&attr);
    /* set the scheduling algorithm to PROCESS or SYSTEM */
    pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
    /* set the scheduling policy - FIFO, RT, or OTHER */
    pthread_attr_setschedpolicy(&attr, SCHED_OTHER);
    /* create the threads */
    . . .
```

56



Solaris Scheduling

Four classes of scheduling

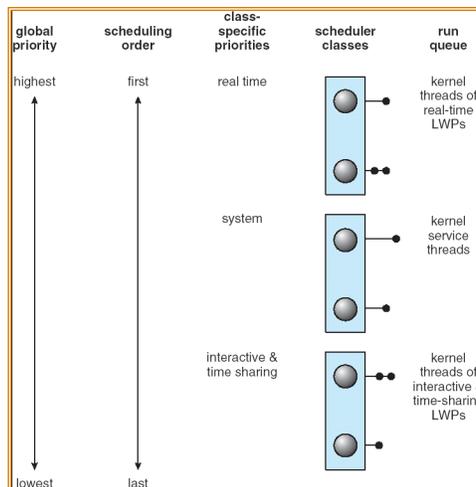
- Real time
- System
- Time sharing
- Interactive

Default class

- time sharing

Algorithm:

- multi-level feedback queue



58

Solaris Scheduling Table (interactive/ time-sharing classes)

Gives high priority for processes returning from sleep, to keep interactive and I/O bound processes busy

Note that high priority processes have short time quanta.

Priorities are lowered when time quantum expires

Low
High

priority	time quantum	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
50	40	40	58
55	40	45	58
59	20	49	59

59

Windows XP Priorities

Preemptive, priority based.

Has **priority classes**, and within each priority class has **relative priorities**.

Except for the real-time class, the priorities are variable.

- Expired time quantum lowers relative priority, returning from sleep-I/O rises it

	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

60

Linux Scheduling

- Preemptive, priority-based
- Two priority ranges: time-sharing and real-time
- Time-sharing
 - Task with highest priority in active array is selected.
 - Task remains in active array as long as time remains in its time slice.
 - When task exhaust time-slice, new priority/timeslice is computed and task is moved to expired array.
 - Priority also affected by interactivity (increases when I/O is long)
 - When active array is empty, expired array becomes active and vice versa
- Real-time
 - Soft real-time
 - Posix.1b compliant – two classes
 - FCFS and RR
 - Highest priority process always runs first

61

Linux Scheduling

Note that the time quanta/priority relationship is opposite, compared to Solaris.

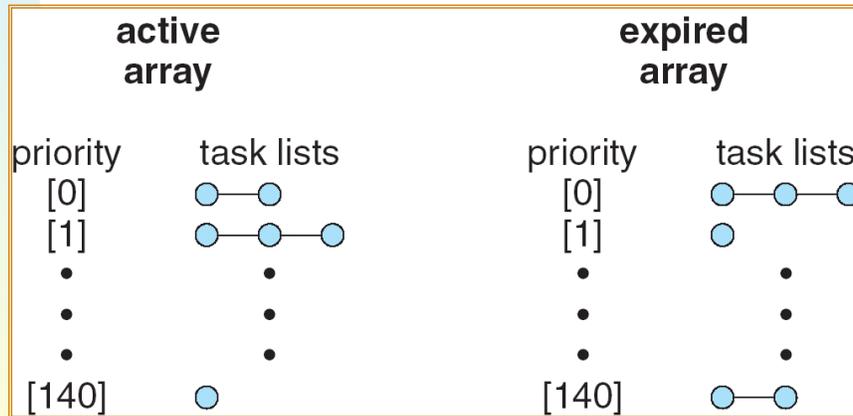
Note also that low numbers indicate high priority.

numeric priority	relative priority		time quantum
0	highest	real-time tasks	200 ms
•			
•			
•			
99		other tasks	
100			
•			
•			
140	lowest		10 ms

62

Linux Scheduling Lists

When active array is empty, move expired array into active array



Algorithm Evaluation

- **Deterministic modeling**
 - takes a particular predetermined workload
 - compute the performance of each algorithm for that workload
 - we did that in class for very small examples
 - was limited applicability
 - how to get representative workloads?
- **Queuing models**
 - Computer system is described as a network of servers, each with its own queue
 - Knowing arrival rates and service rates, we can compute utilizations, average queue lengths, average wait time...

Algorithm Evaluation

Simulation

- Create simulator, run different scheduling algorithms and evaluate results
- How to feed it the data about CPU and I/O bursts?
 - Generate randomly based on some empirically observed distribution
 - Feed it data from a real system – trace tapes
- Still, the results are not always representative

Implement & test in real life

- High cost
- Still, the use of the system might change over time, and with it also the requirements

65

Algorithm Evaluation

- How to select CPU-scheduling algorithm for an OS?
- Determine criteria, then evaluate algorithms
- **Deterministic modeling**
 - Type of **analytic evaluation**
 - Takes a particular predetermined workload and defines the performance of each algorithm for that workload
- Consider 5 processes arriving at time 0:

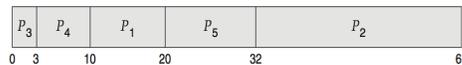
Process	Burst Time
P_1	10
P_2	29
P_3	3
P_4	7
P_5	12

Deterministic Evaluation

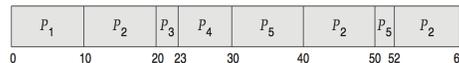
- For each algorithm, calculate minimum average waiting time
- Simple and fast, but requires exact numbers for input, applies only to those inputs



- **FCS is 28ms:**



- **Non-preemptive SFJ is 13ms:**



- **RR is 23ms:**

Queueing Models

- Describes the arrival of processes, and CPU and I/O bursts probabilistically
 - Commonly exponential, and described by mean
 - Computes average throughput, utilization, waiting time, etc
- Computer system described as network of servers, each with queue of waiting processes
 - Knowing arrival rates and service rates
 - Computes utilization, average queue length, average wait time, etc

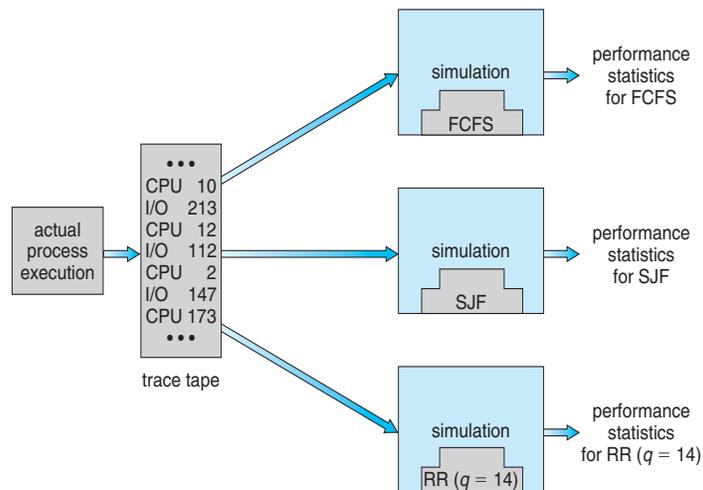
Little' s Formula

- n = average queue length
- W = average waiting time in queue
- λ = average arrival rate into queue
- Little' s law – in steady state, processes leaving queue must equal processes arriving, thus
 $n = \lambda \times W$
 - Valid for any scheduling algorithm and arrival distribution
- For example, if on average 7 processes arrive per second, and normally 14 processes in queue, then average wait time per process = 2 seconds

Simulations

- Queueing models limited
- **Simulations** more accurate
 - Programmed model of computer system
 - Clock is a variable
 - Gather statistics indicating algorithm performance
 - Data to drive simulation gathered via
 - Random number generator according to probabilities
 - Distributions defined mathematically or empirically
 - Trace tapes record sequences of real events in real systems

Evaluation of CPU Schedulers by Simulation



Implementation

- Even simulations have limited accuracy
- Just implement new scheduler and test in real systems
 - High cost, high risk
 - Environments vary
- Most flexible schedulers can be modified per-site or per-system
 - Or APIs to modify priorities
 - But again environments vary