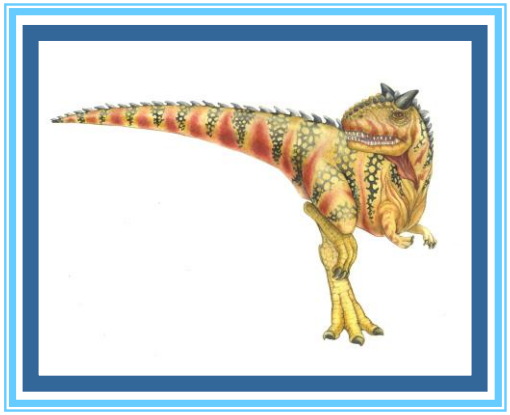
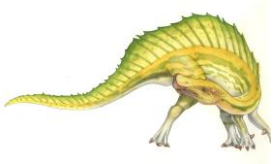
Chapter 5: CPU Scheduling



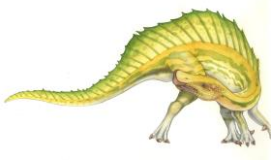
Outline

- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Thread Scheduling
- Multi-Processor Scheduling
- Real-Time CPU Scheduling
- Operating Systems Examples
- Algorithm Evaluation



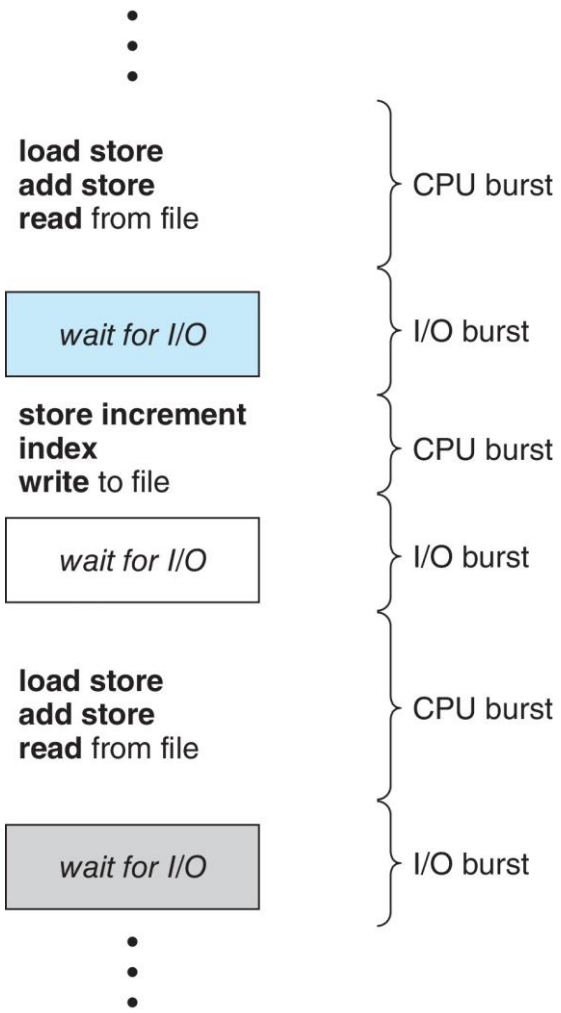
Objectives

- Describe various CPU scheduling algorithms
- Assess CPU scheduling algorithms based on scheduling criteria
- Explain the issues related to multiprocessor and multicore scheduling
- Describe various real-time scheduling algorithms
- Describe the scheduling algorithms used in the Windows, Linux, and Solaris operating systems
- Apply modeling and simulations to evaluate CPU scheduling algorithms



Basic Concepts

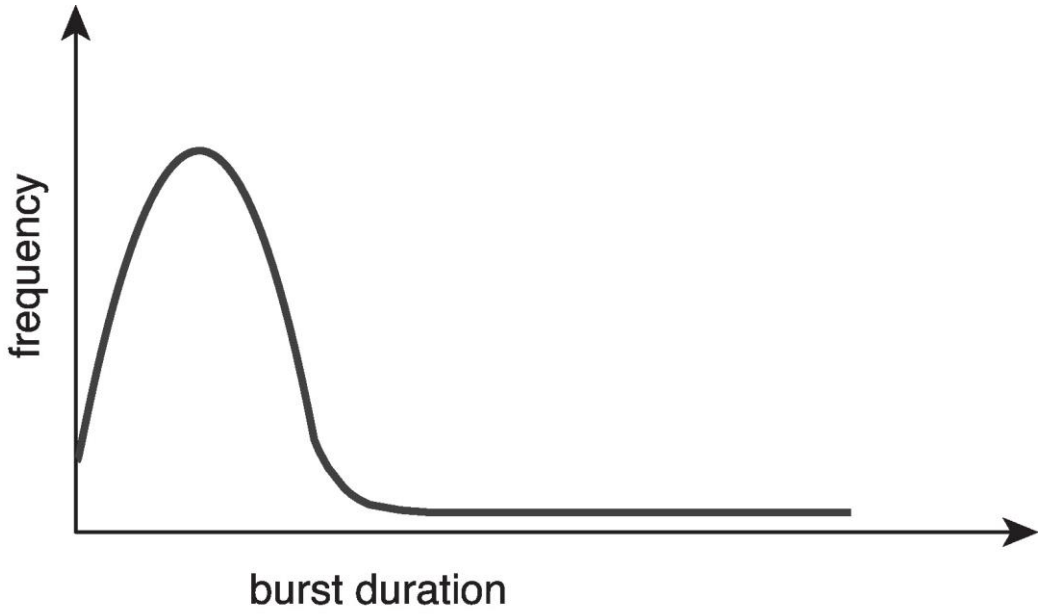
- Maximum CPU utilization obtained with multiprogramming
- CPU–I/O Burst Cycle – Process execution consists of a **cycle** of CPU execution and I/O wait
- **CPU burst** followed by **I/O burst**
- CPU burst distribution is of main concern

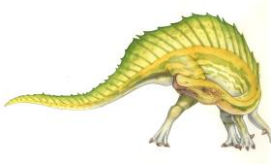


Histogram of CPU-burst Times

Large number of short bursts

Small number of longer bursts





CPU Scheduler

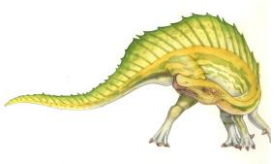
- The **CPU scheduler** selects from among the processes in ready queue, and allocates a CPU core to one of them
  - Queue may be ordered in various ways
- CPU scheduling decisions may take place when a process:
  1. Switches from running to waiting state
  2. Switches from running to ready state
  3. Switches from waiting to ready
  4. Terminates
- For situations 1 and 4, there is no choice in terms of scheduling. A new process (if one exists in the ready queue) must be selected for execution.
- For situations 2 and 3, however, there is a choice.

Preemptive and Nonpreemptive Scheduling

- When scheduling takes place only under circumstances 1 and 4, the scheduling scheme is **nonpreemptive**.
- Otherwise, it is **preemptive**.
- Under Nonpreemptive scheduling, once the CPU has been allocated to a process, the process keeps the CPU until it releases it either by terminating or by switching to the waiting state.
- Virtually all modern operating systems including Windows, MacOS, Linux, and UNIX use preemptive scheduling algorithms.

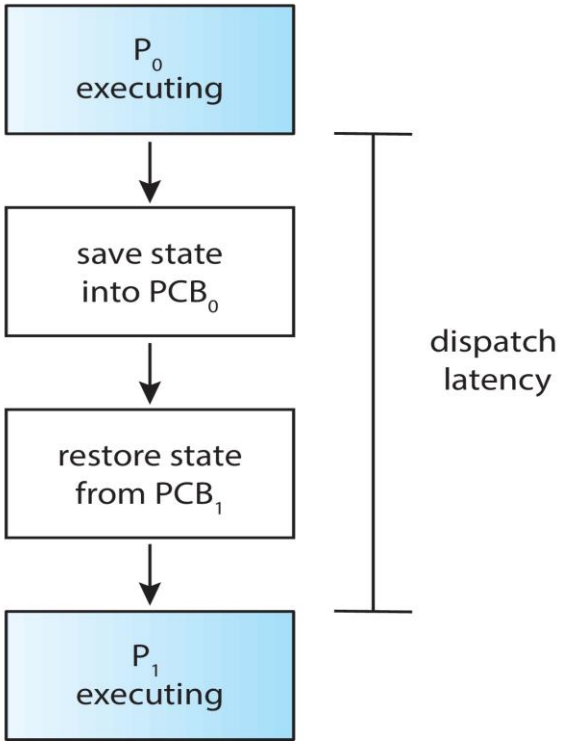
Preemptive Scheduling and Race Conditions

- Preemptive scheduling can result in race conditions when data are shared among several processes.
- Consider the case of two processes that share data. While one process is updating the data, it is preempted so that the second process can run. The second process then tries to read the data, which are in an inconsistent state.
- This issue will be explored in detail in Chapter 6.



Dispatcher

- Dispatcher module gives control of the CPU to the process selected by the CPU 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

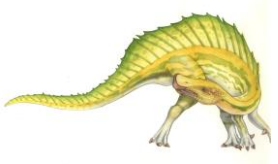


Scheduling Criteria

- **CPU utilization** – keep the CPU as busy as possible
- **Throughput** – # of processes that complete their execution per time unit
- **Turnaround time** – amount of time to execute a particular process
- **Waiting time** – amount of time a process has been waiting in the ready queue
- **Response time** – amount of time it takes from when a request was submitted until the first response is produced.

Scheduling Algorithm Optimization Criteria

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



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:

$$P_2 , P_3 , P_1$$

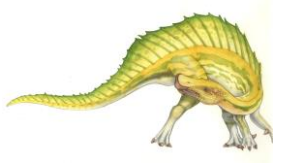
- The Gantt chart for the schedule is:

- 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
  - Consider one CPU-bound and many I/O-bound processes

Shortest-Job-First (SJF) Scheduling

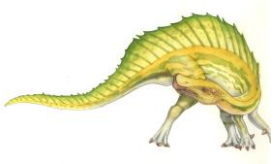
- initial folosit pt sisteme batch, cand timpii de rulare sunt cunoscuti a priori
- se alege cel mai scurt job primul
- SJF e optimal
  - Pp t_1, t_2, … t_n timpii de rulare
  - Job 1 termina la t_1
  - Job 2 termina la $t_1 + t_2$
  - …
  - Job n termina la $t_1 + t_2 + \ldots + t_n$

  $\Rightarrow$ timp mediu de asteptare = $\frac{1}{n}(n\ t_1 + (n-1)\ t_2 + \ldots\ t_n)$
- contributia lui t_1 e cea mai mare => valoarea lui t_1 trebuie sa fie cea mai mica pt a obtine valoarea minima a timpului mediu de asteptare
- similar, t_2 e urmatorul cel mai mic “contributor”, samd



Shortest-Job-First (SJF) Scheduling

- Obs: *SJF e optimal doar daca joburile sunt disponibile simultan !*
- contraexemplu:
  - 5 joburi A, B, C, D, E cu duratele 2,4,1,1,1 si timpi de sosire 0,0,3,3,3
  - SJF impune ordinea A, B, C, D, E => TA mediu = 4.6
  - in schimb, o planificare B, C, D, E, A => TA mediu = 4.4



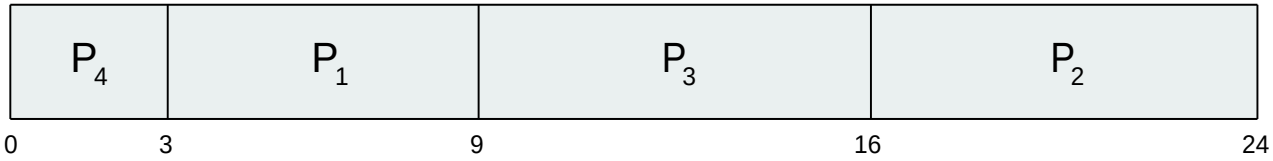
Shortest-Job-First (SJF) Scheduling

- Associate with each process the length of its next CPU burst
  - Use these lengths to schedule the process with the shortest time
- SJF is optimal – gives minimum average waiting time for a given set of processes
- Preemptive version called **shortest-remaining-time-first**
- How do we determine the length of the next CPU burst?
  - Could ask the user
  - Estimate

Example of SJF

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

- SJF scheduling chart



- Average waiting time = (3 + 16 + 9 + 0) / 4 = 7

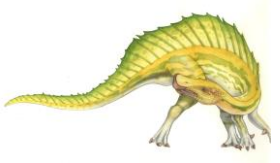
Determining Length of Next CPU Burst

- Can only estimate the length – should be similar to the previous one
  - Then pick process with shortest predicted next CPU burst
- Can be done by using the length of previous CPU bursts, using exponential averaging

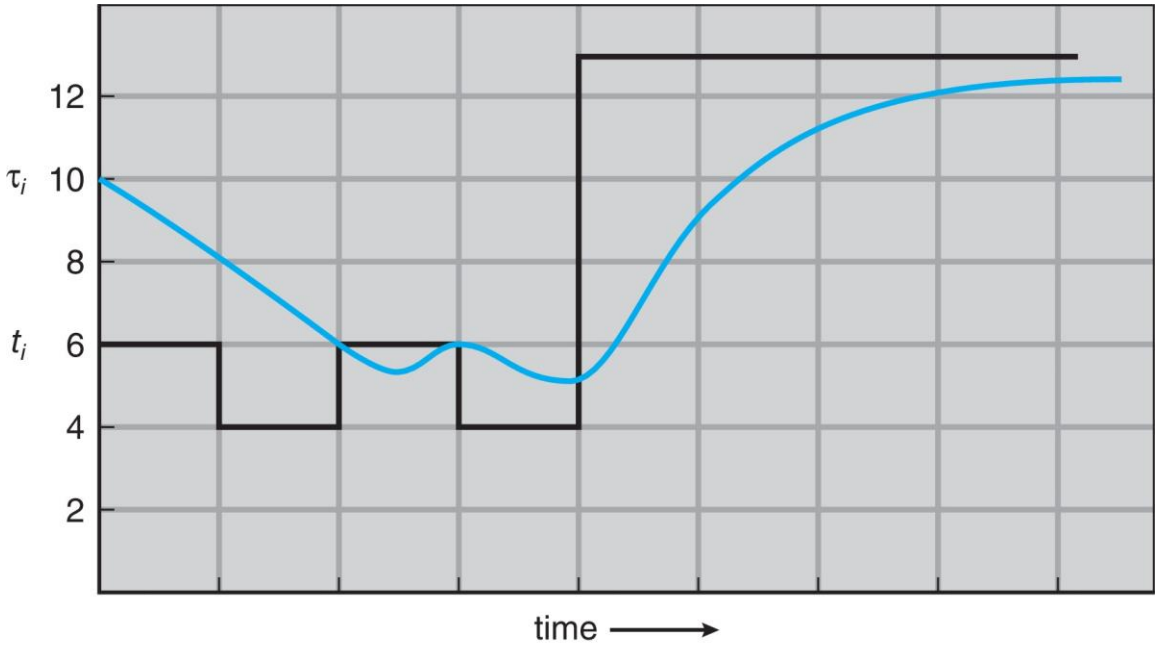
  1. t_n = actual length of n^{th} CPU burst
  2. τ_{n+1} = predicted value for the next CPU burst
  3. α, $0 \leq \alpha \leq 1$
  4. Define:

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

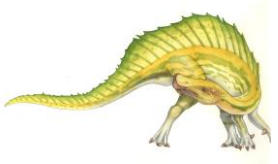
- Commonly, α set to ½



Prediction of the Length of the Next CPU Burst



CPU burst (t_i)	6	4	6	4	13	13	13	...
"guess" (τ_i)	10	8	6	6	5	9	11	12	...



Examples of Exponential Averaging

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

$$\tau_{n+1} = \alpha\, t_n + (1 - \alpha)\alpha\, t_{n-1} + \ldots$$
$$+ (1 - \alpha)^j \alpha\, t_{n-j} + \ldots$$
$$+ (1 - \alpha)^{n+1} \tau_0$$

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

Shortest Remaining Time First Scheduling

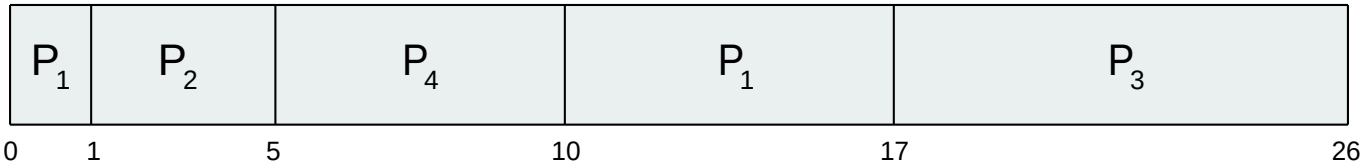
- Preemptive version of SJN
- Whenever a new process arrives in the ready queue, the decision on which process to schedule next is redone using the SJN algorithm.
- Is SRT more “optimal” than SJN in terms of the minimum average waiting time for a given set of processes?

Example of Shortest-remaining-time-first

- Now we add the concepts of varying arrival times and preemption to the analysis

Process	*Arrival* Time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

- *Preemptive* SJF Gantt Chart



- Average waiting time = [(10-1)+(1-1)+(17-2)+(5-3)]/4 = 26/4 = 6.5
- SJF nepreemptiv are average waiting time = 7.75 ms

Round Robin (RR)

- Each process gets a small unit of CPU time (**time quantum** *q*), 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.
- Timer interrupts every quantum to schedule next process
- Performance
  - *q* large $\Rightarrow$ FIFO (FCFS)
  - *q* small $\Rightarrow$ RR
- Note that q must be large with respect to context switch, otherwise overhead is too high

Observatii RR

- run/ready queue e implementata circular, mutarea procesului la sf. listei revine la a muta un pointer circular
- alegerea duratei cuantei de timp
  - dc. context-switch-ul dureaza mult (ex. 5 ms) fata de durata cuantei (ex. 20 ms) => 20 ms de lucru util urmat de 5 ms context switch => 20% overhead administrativ !

  $\Rightarrow$ e bine sa alegem cuanta >> durata context switch-ului

  Q: cat de mare?

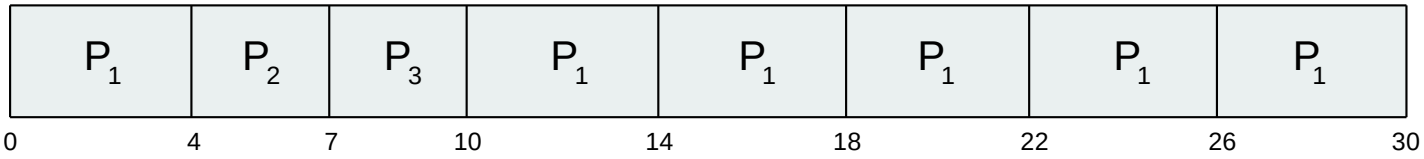
  Pp. q = 500ms si 10 useri interactivi care lanseaza simultan 10 programe => primul user vede imediat o parte din rezultate, al doilea dupa ½ s, samd Al 10-lea asteapta 5 s !!!
- concluzie: - cuanta scurta implica multe context switch-uri si reduce eficienta utilizarii CPU

  - cuanta prea mare reduce timpul de raspuns pt programele interactive

Example of RR with Time Quantum = 4

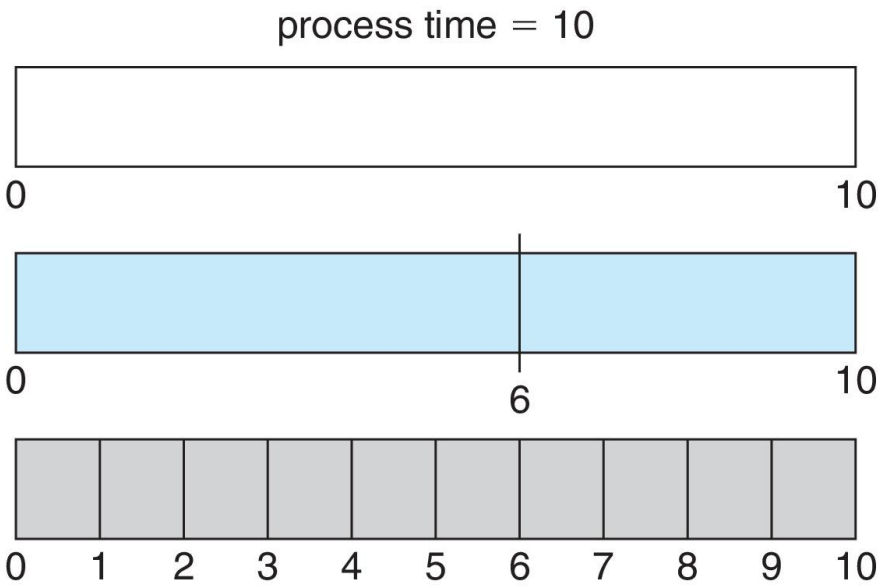
Process	Burst Time
P_1	24
P_2	3
P_3	3

- The Gantt chart is:



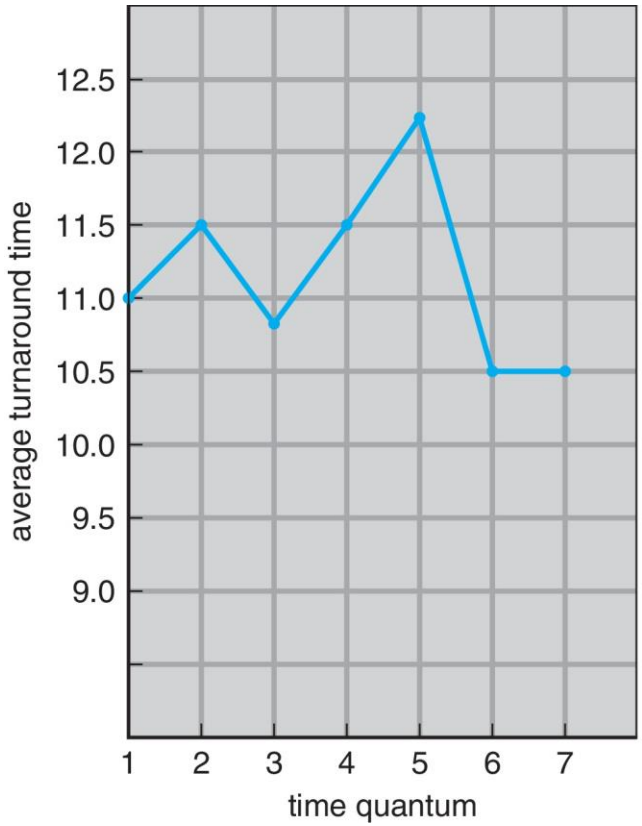
- Average waiting time = 17/3 = 5.66 ms
- Typically, higher average turnaround than SJF, but better ***response***
- q should be large compared to context switch time
  - q usually 10 milliseconds to 100 milliseconds,
  - Context switch < 10 microseconds

Time Quantum and Context Switch Time



quantum	context switches
12	0
6	1
1	9

Turnaround Time Varies With The Time Quantum



process	time
P_1	6
P_2	3
P_3	1
P_4	7

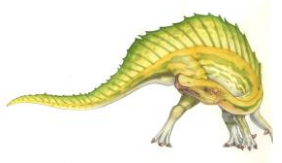
80% of CPU bursts should be shorter than q

Planificarea cu prioritati

- RR pp. ca toate procesele au aceeasi importanta, nerealist
- cuantificarea “importantei” se face cu ajutorul prioritatilor
- procesul cel mai important are prioritate maxima
- planificatorul alege intotdeauna procesul cu prioritate maxima
- pentru a impiedica procesele cu prioritate maxima sa ruleze indefinit, planificatorul poate reduce prioritatea procesului care ruleaza cu fiecare tact de ceas
- cand prioritatea scade sub cea a urmatorului proces prioritar => context switch
- prioritatile se pot asigna fie static fie dinamic
- asignare dinamica:
  - procesele I/O-bound ruleaza putin, e bine sa primeasca imediat CPU pt a lansa cererea de I/O urmatoare

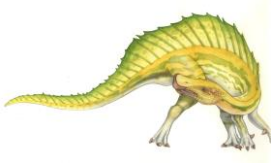
Planificarea cu prioritati

- asignare dinamica:
  - procesele I/O-bound ruleaza putin, e bine sa primeasca imediat CPU pt a lansa cererea de I/O urmatoare
  - algoritm:
    - procesul I/O-bound primeste prioritatea *1 / f*
    - f = fractiunea din cuanta CPU utilizata la ultima rulare de catre proces => cu cat ruleaza mai putin, cu are prioritate mai mare
- clase de prioritate + RR:
  - *n* clase de prioritate
  - se ruleaza RR procesele din clasa de prioritate maxima
  - daca nu exista procese in clasa de prioritate maxima, se trece la clasa urmatoare de prioritate
  - si aici se ajusteaza prioritatile pt a evita starvation in clasele de prioritate mica



Priority Scheduling

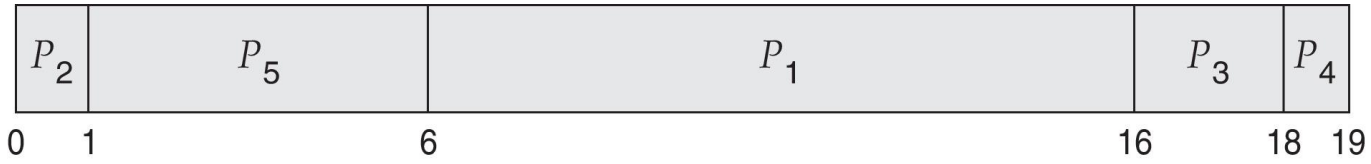
- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer ≡ highest priority)
  - Preemptive
  - Nonpreemptive
- SJF is priority scheduling where priority is the inverse of predicted next CPU burst time
- Problem ≡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



Example of Priority Scheduling

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

- Priority scheduling Gantt Chart



- Average waiting time = 8.2

Priority Scheduling w/ Round-Robin

- Run the process with the highest priority. Processes with the same priority run round-robin
- Example:

Process	Burst Time	Priority
P_1	4	3
P_2	5	2
P_3	8	2
P_4	7	1
P_5	3	3

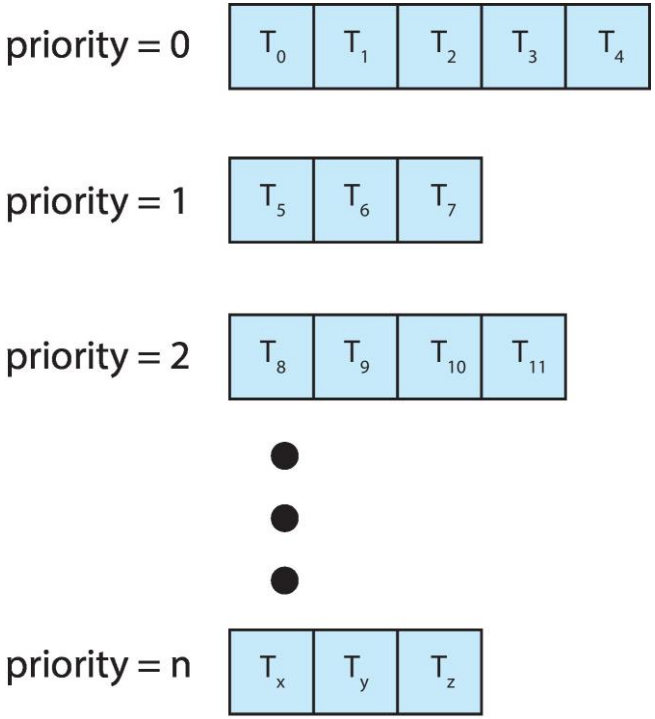
- Gantt Chart with time quantum = 2

Multilevel Queue

- The ready queue consists of multiple queues
- Multilevel queue scheduler defined by the following parameters:
  - Number of queues
  - Scheduling algorithms for each queue
  - Method used to determine which queue a process will enter when that process needs service
  - Scheduling among the queues

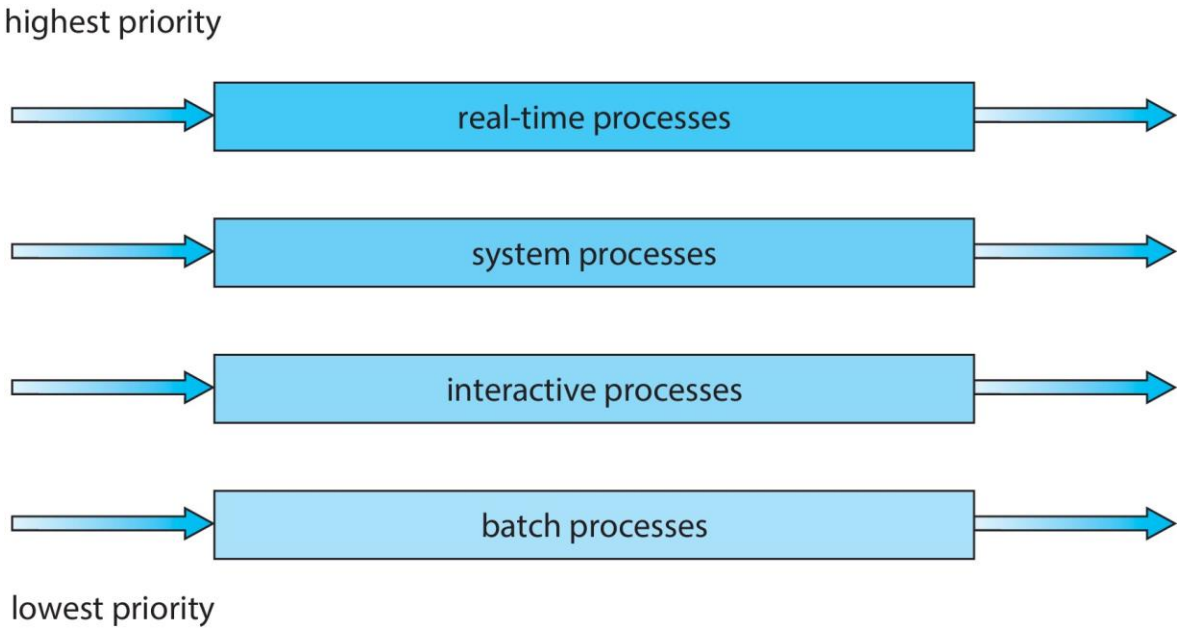
Multilevel Queue

- With priority scheduling, have separate queues for each priority.
- Schedule the process in the highest-priority queue!



Multilevel Queue

- Prioritization based upon process type

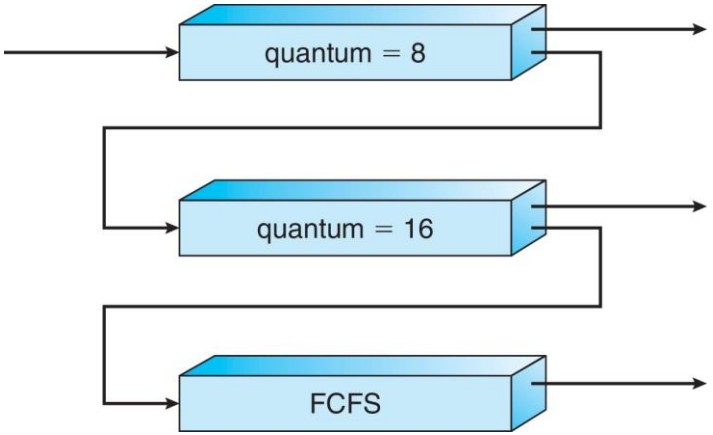


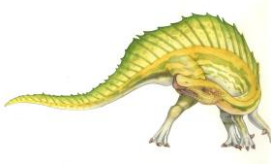
Multilevel Feedback Queue

- A process can move between the various queues.
- 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
- Aging can be implemented using multilevel feedback queue

Example of Multilevel Feedback Queue

- Three queues:
  - Q_0 – RR with time quantum 8 milliseconds
  - Q_1 – RR time quantum 16 milliseconds
  - Q_2 – FCFS
- Scheduling
  - A new process enters queue Q_0 which is served in RR
    - When it gains CPU, the process receives 8 milliseconds
    - If it does not finish in 8 milliseconds, the process is moved to queue Q_1
  - At Q_1 job is again served in RR and receives 16 additional milliseconds
    - If it still does not complete, it is preempted and moved to queue Q_2





Thread Scheduling

- Distinction between user-level and kernel-level threads
- When kernel threads supported, kernel threads scheduled, not processes
- Many-to-one and many-to-many models, thread library schedules user-level threads to run on LWP
  - Known as **process-contention scope** (**PCS**) since scheduling competition is within the process
  - Typically done via priority set by programmer
- Kernel thread scheduled onto available CPU is **system-contention scope** (**SCS**) – competition among all threads in system

Pthread Scheduling

- API allows specifying either PCS or SCS during thread creation
  - PTHREAD_SCOPE_PROCESS schedules threads using PCS scheduling
  - PTHREAD_SCOPE_SYSTEM schedules threads using SCS scheduling
- Can be limited by OS – Linux and macOS only allow PTHREAD_SCOPE_SYSTEM

Pthread Scheduling API

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5
int main(int argc, char *argv[]) {
   int i, scope;
   pthread_t tid[NUM THREADS];
   pthread_attr_t attr;
   /* get the default attributes */
   pthread_attr_init(&attr);
   /* first inquire on the current scope */
   if (pthread_attr_getscope(&attr, &scope) != 0)
      fprintf(stderr, "Unable to get scheduling scope\n");
   else {
      if (scope == PTHREAD_SCOPE_PROCESS)
         printf("PTHREAD_SCOPE_PROCESS");
      else if (scope == PTHREAD_SCOPE_SYSTEM)
         printf("PTHREAD_SCOPE_SYSTEM");
      else
         fprintf(stderr, "Illegal scope value.\n");
   }
```

Pthread Scheduling API

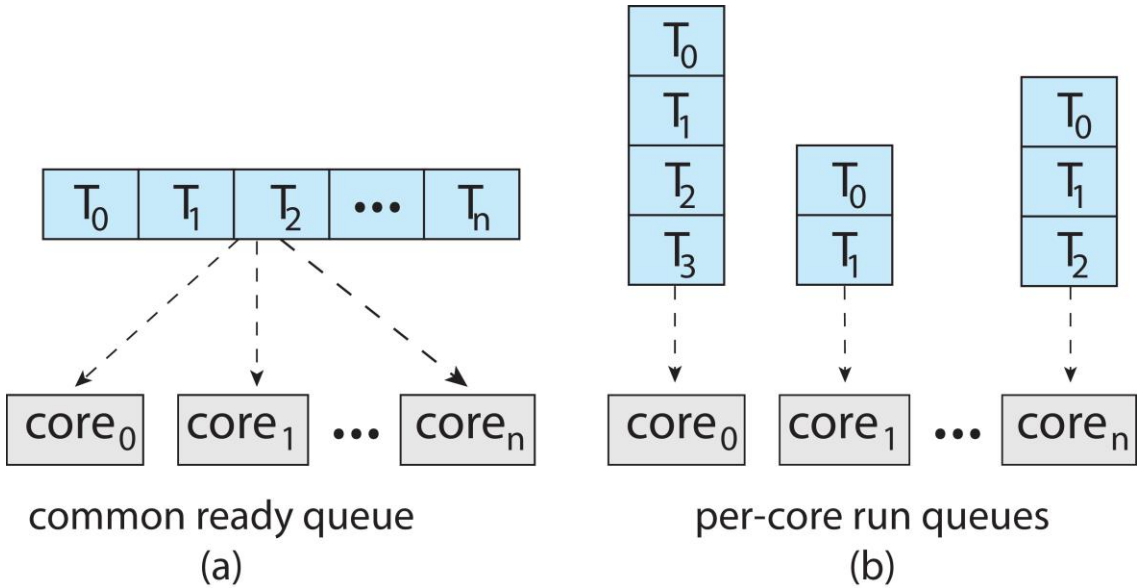
```
   /* set the scheduling algorithm to PCS or SCS */
   pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
   /* create the threads */
   for (i = 0; i < NUM_THREADS; i++)
      pthread_create(&tid[i],&attr,runner,NULL);
   /* now join on each thread */
   for (i = 0; i < NUM_THREADS; i++)
      pthread_join(tid[i], NULL);
}
/* Each thread will begin control in this function */
void *runner(void *param)
{
   /* do some work ... */
   pthread_exit(0);
}
```

Multiple-Processor Scheduling

- CPU scheduling more complex when multiple CPUs are available
- Multiprocess may be any one of the following architectures:
  - Multicore CPUs
  - Multithreaded cores
  - NUMA systems
  - Heterogeneous multiprocessing

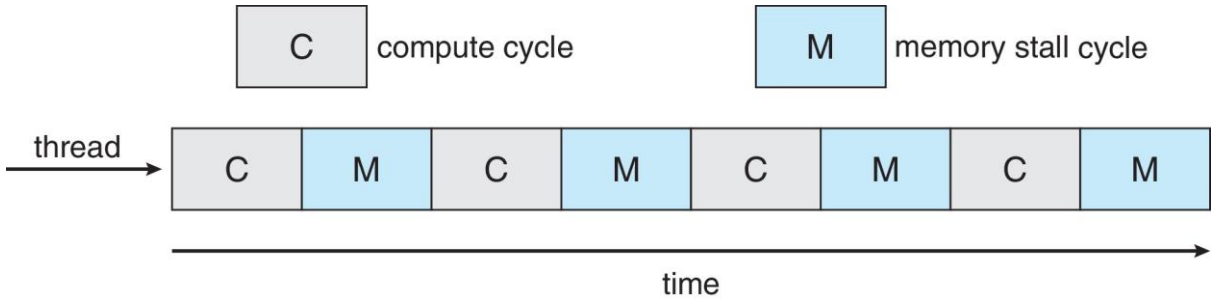
Multiple-Processor Scheduling

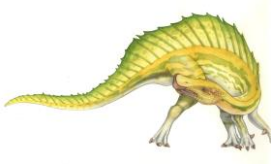
- Symmetric multiprocessing (SMP) is where each processor is self scheduling.
- All threads may be in a common ready queue (a)
- Each processor may have its own private queue of threads (b)



Multicore Processors

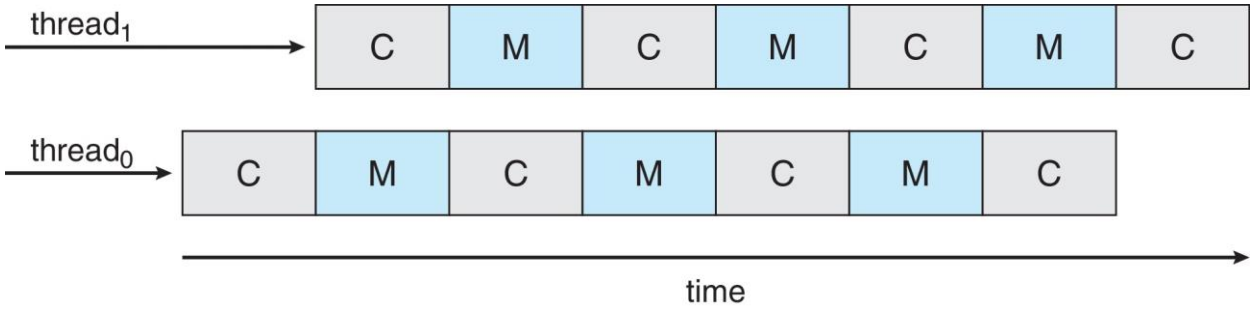
- Recent trend to place multiple processor cores on same physical chip
- Faster and consumes less power
- Multiple threads per core also growing
  - Takes advantage of memory stall to make progress on another thread while memory retrieve happens
- Figure





Multithreaded Multicore System

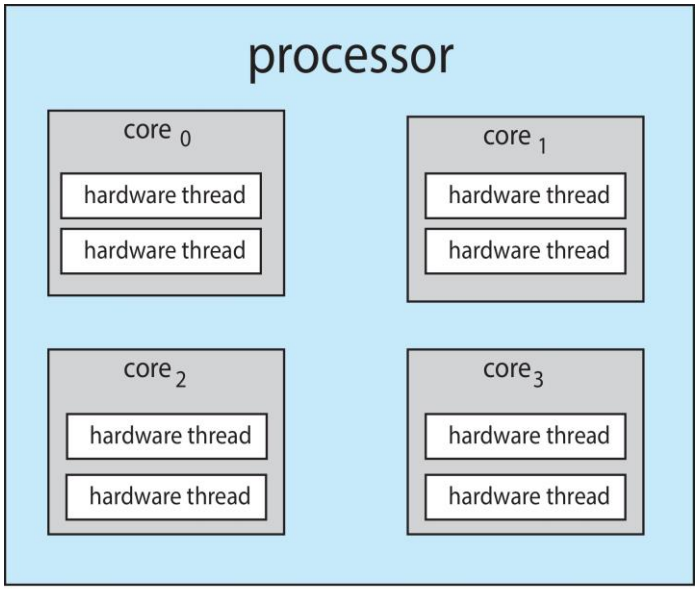
- Each core has > 1 hardware threads.
- If one thread has a memory stall, switch to another thread!
- Figure

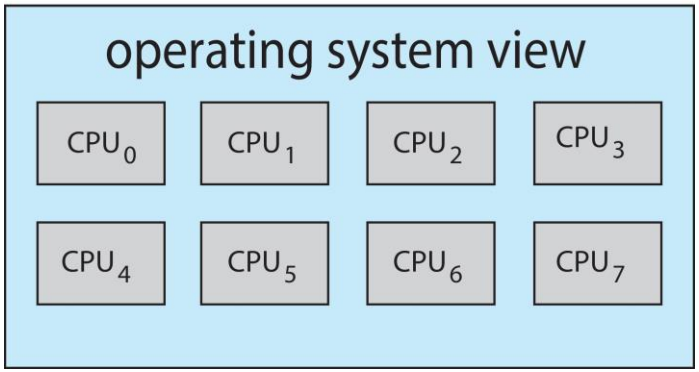


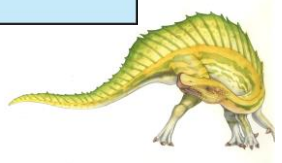
Multithreaded Multicore System

- **Chip-multithreading** (CMT) assigns each core multiple hardware threads. (Intel refers to this as **hyperthreading**.)

- On a quad-core system with 2 hardware threads per core, the operating system sees 8 logical processors.

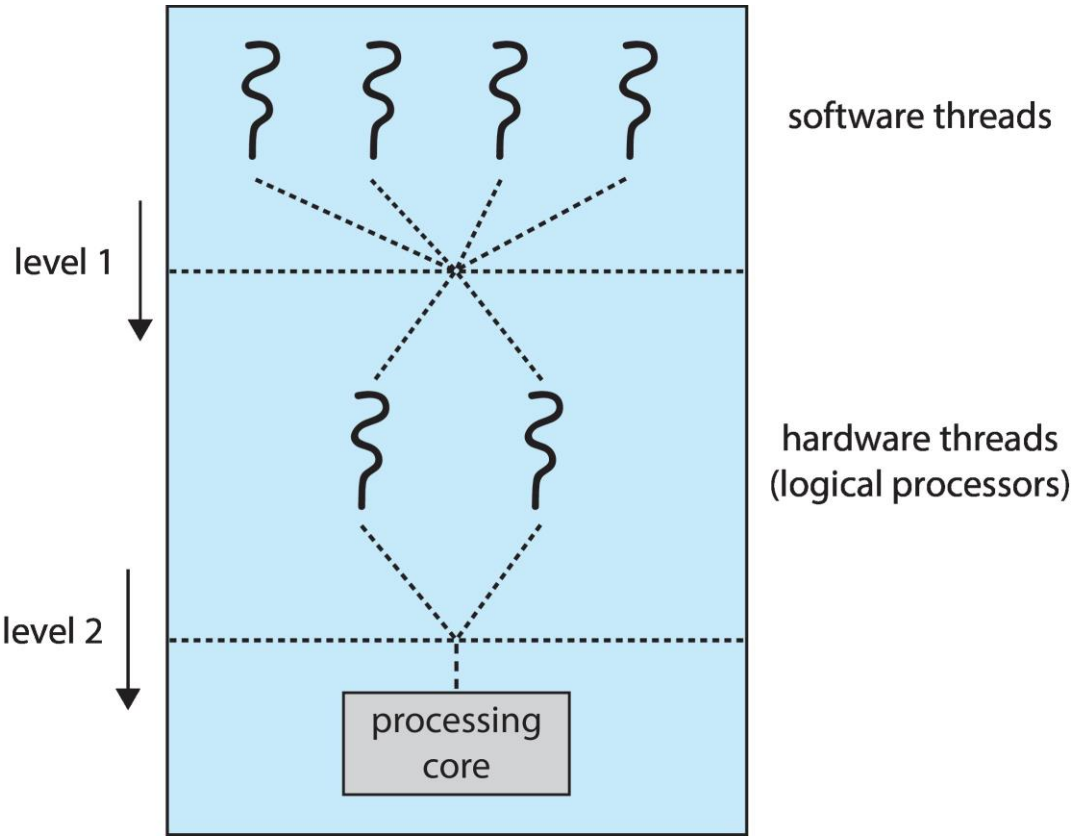






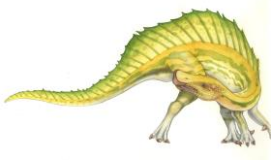
Multithreaded Multicore System

- Two levels of scheduling:
  1. The operating system deciding which software thread to run on a logical CPU
  2. How each core decides which hardware thread to run on the physical core.



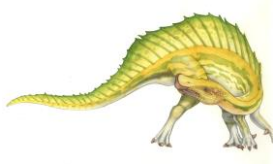
Multiple-Processor Scheduling – Load Balancing

- If SMP, need to keep all CPUs loaded for efficiency
- **Load balancing** attempts to keep workload evenly distributed
- **Push migration** – periodic task checks load on each processor, and if found pushes task from overloaded CPU to other CPUs
- **Pull migration** – idle processors pulls waiting task from busy processor



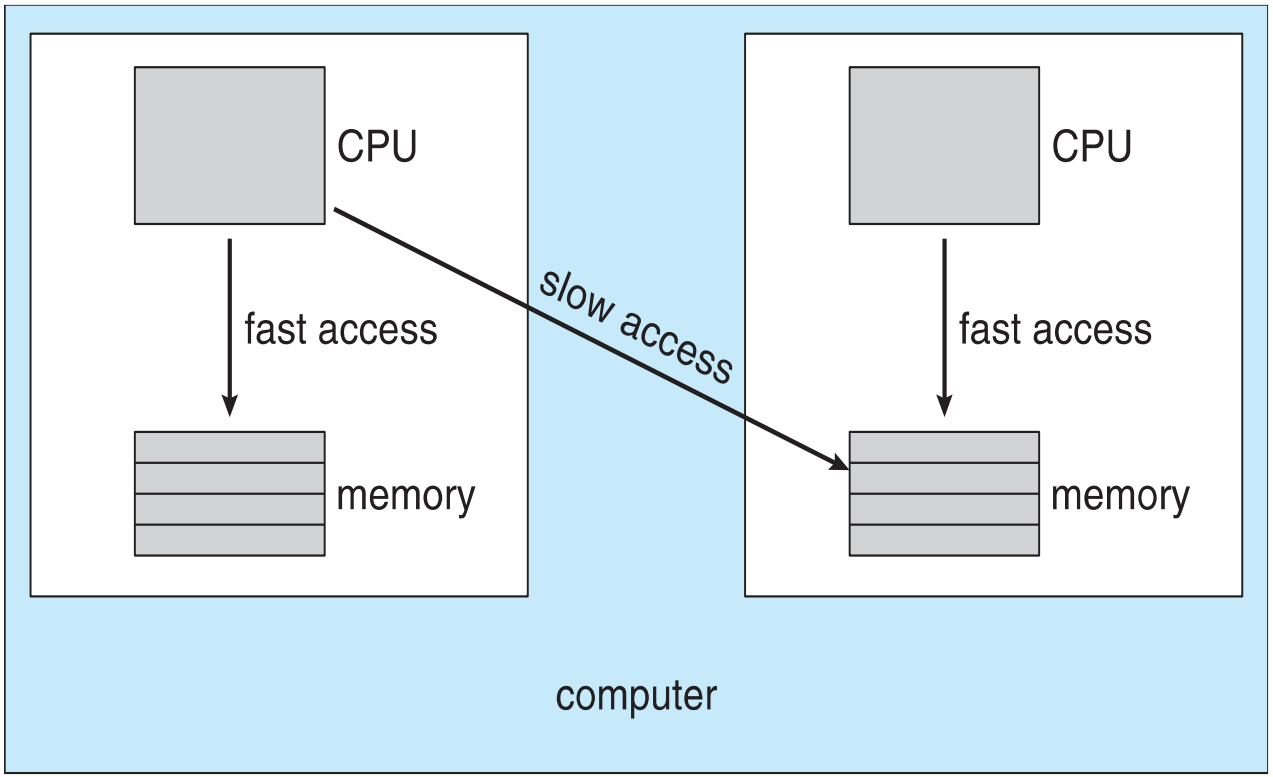
Multiple-Processor Scheduling – Processor Affinity

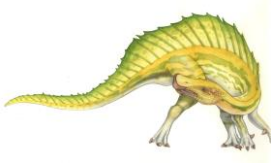
- When a thread has been running on one processor, the cache contents of that processor stores the memory accesses by that thread.
- We refer to this as a thread having affinity for a processor (i.e., “processor affinity”)
- Load balancing may affect processor affinity as a thread may be moved from one processor to another to balance loads, yet that thread loses the contents of what it had in the cache of the processor it was moved off of.
- **Soft affinity** – the operating system attempts to keep a thread running on the same processor, but no guarantees.
- **Hard affinity** – allows a process to specify a set of processors it may run on.



NUMA and CPU Scheduling

If the operating system is **NUMA-aware**, it will assign memory closes to the CPU the thread is running on.





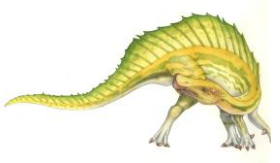
NORMA Scheduling

- in sistemele distribuite, fiecare procesor executa local algoritmul de planificare
- Q: cum se comporta un grup de procese care interactioneaza strans si ruleaza pe masini diferite?
- ex: procesele A, B ruleaza pe P1; C si D pe P2
- masinile P1 si P2 ruleaza RR cu q=10ms
- A si C ruleaza in cuantele pare, B si D in cele impare
- A porneste si cheama D in cuanta para, D nu ruleaza (C ruleaza in cuantele pare)
- dupa 10ms, D primeste mesajul lui A si raspunde; cuanta e impara, A nu ruleaza (B ruleaza)
- dupa inca 10 ms, A primeste mesajul lui D

$\Rightarrow$ schimbul de mesaje are un overhead de 20 ms

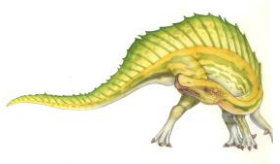
Co-scheduling

- idee: procesele care comunica frecvent sa ruleze simultan pe masini diferite
- implementare co-scheduling cu matrici
  - fiecare coloana contine procesele rulate pe un anume procesor (masina)
  - fiecare linie contine procesele care ruleaza simultan procesoare
  - i.e., coloanele reprezinta procesoare(masini), liniile cuante de timp
  - fiecare procesor planifica RR local
  - procesoarele isi sincronizeaza cuantele RR folosind bcast (un ceas sincron ca la SIMD, posibil implementat prin NTP)
  - co-scheduling: toate procesele unui grup au alocate aceleasi cuante de timp (i.e., apar pe aceleasi linii ale matricii in diferite coloane) => maximizarea paralelismului disponibil si a throughput-ului comunicatiei



Planificare de timp real

- *sistem de timp real* = sistem in care corectitudinea executiei unui program nu depinde doar de corectitudinea rezultatelor calculate si si de **timpul** la care sunt livrate rezultatele
  - **hard** – toate rezultatele trebuie livrate la timp
  - **soft** – rezultatele calculate pot fi acceptate si cu intarziere
- definitii
  - task = o singura executie a unui bloc de cod
  - timp de sosire al taskului = timpul la care sistemul devine constient ca taskul trebuie executat
- taskuri *periodice*
  - sosesc la intervale regulate de timp
  - T(t) = timp fix inter-sosiri (perioada)
  - C(t) = timpul de calcul

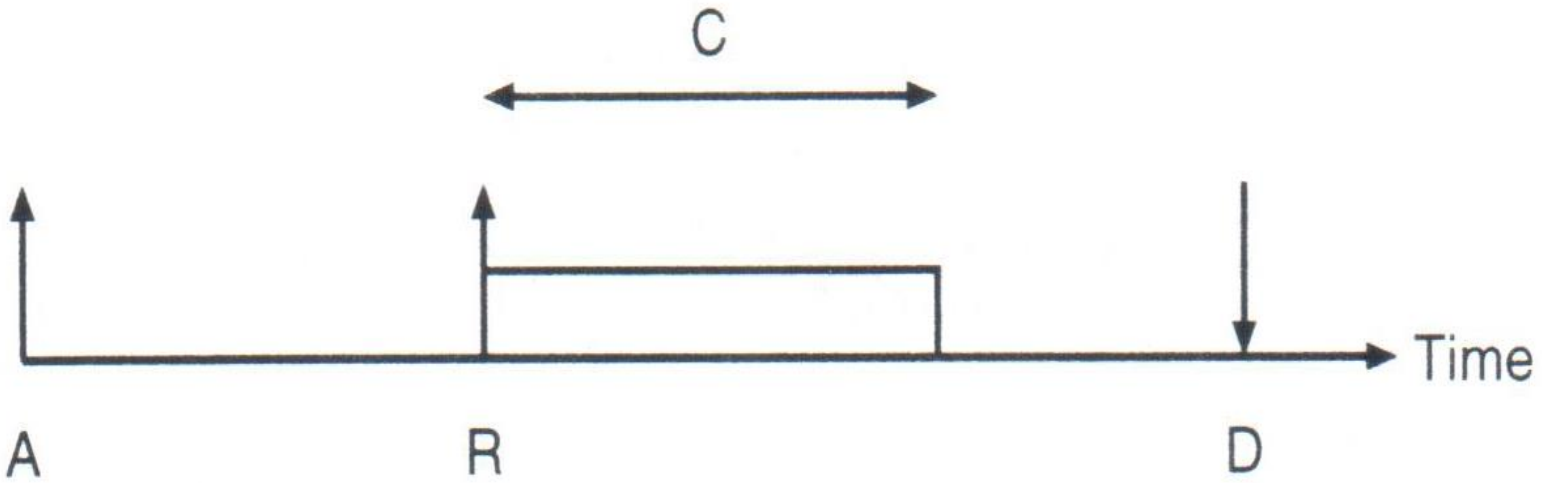


Planificare de timp real

- taskuri *aperiodice*
  - caracterizate de rate de sosiri stochastice
  - pot veni “in rafala” => mai multe instante ale unui task aperiodic pot veni intr-un interval scurt de timp => supraincarcare sistem
  - in consecinta, se pp. ca exista un timp minim intre sosiri succesive ale aceluiasi task
  - asemenea taskuri s.n. *sporadice*
  - si taskurile sporadice pot veni intr-un interval scurt de timp; daca sistemul n-are suficiente resurse disponibile se ajunge la nerespectarea deadline-urilor
  - se spune ca sistemul se afla in stare de *supraincarcare trecatoare* (“transient overload”)

Planificare de timp real

- timpul de eliberare (“release time”) = timpul la care taskului i se permite sa porneasca executia
- deadline = timpul pana la care trebuie sa se termine executia taskului
- diagrama de timp
  - S -> R -> C -> D (sosire, release, calcul, deadline)

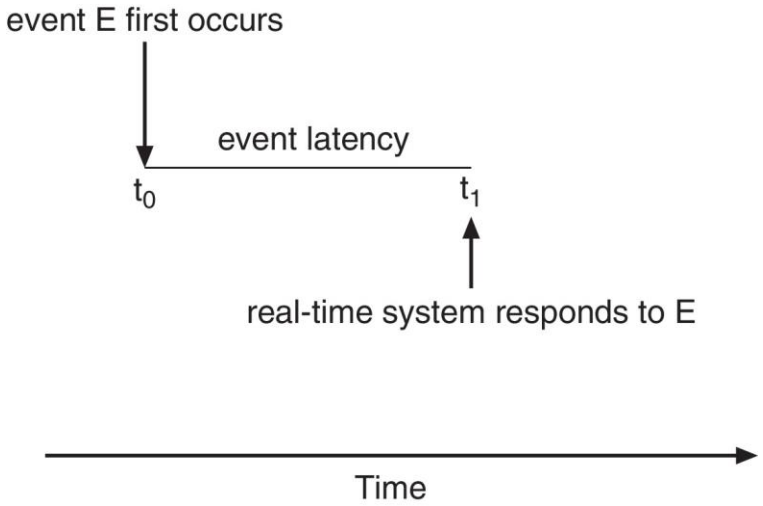


Real-Time CPU Scheduling

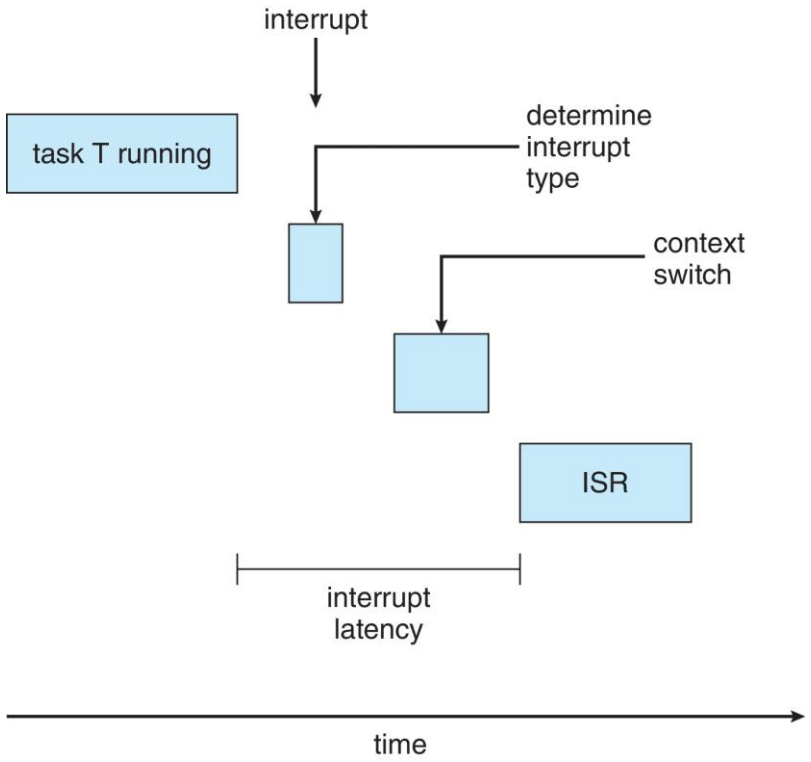
- Can present obvious challenges
- **Soft real-time systems** – Critical real-time tasks have the highest priority, but no guarantee as to when tasks will be scheduled
- **Hard real-time systems – task must be serviced by its deadline**

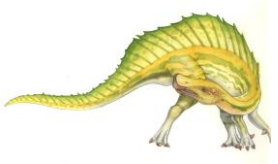
Real-Time CPU Scheduling

- Event latency – the amount of time that elapses from when an event occurs to when it is serviced.
- Two types of latencies affect performance
  1. **Interrupt latency** – time from arrival of interrupt to start of routine that services interrupt
  2. **Dispatch latency** – time for schedule to take current process off CPU and switch to another



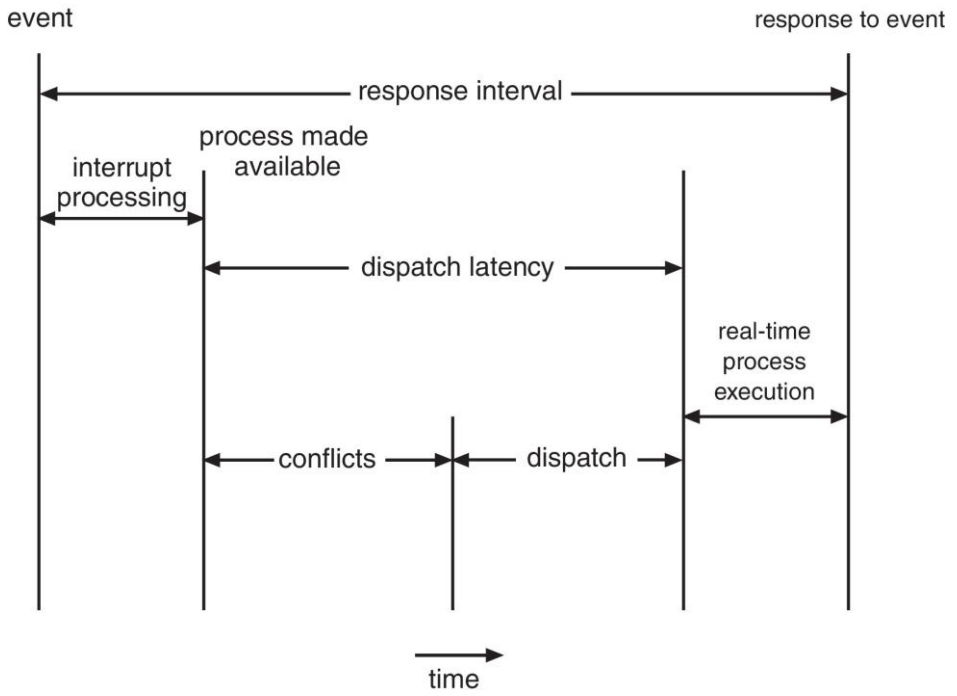
Interrupt Latency

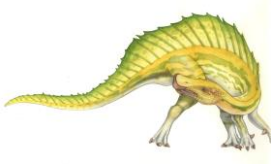




Dispatch Latency

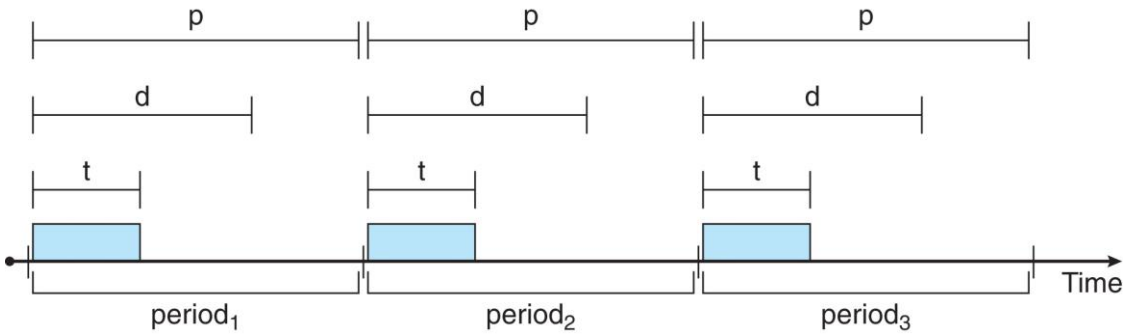
- Conflict phase of dispatch latency:
  1. Preemption of any process running in kernel mode
  2. Release by low-priority process of resources needed by high-priority processes





Priority-based Scheduling

- For real-time scheduling, scheduler must support preemptive, priority-based scheduling
  - But only guarantees soft real-time
- For hard real-time must also provide ability to meet deadlines
- Processes have new characteristics: **periodic** ones require CPU at constant intervals
  - Has processing time *t*, deadline *d,* period *p*
  - $0 \leq t \leq d \leq p$
  - **Rate** of periodic task is $1/p$



Rate Monotonic (RM)

- algoritm pt taskuri periodice (interesante pt modelarea usoara si folosirea pe scara larga in industrie/automatizari)
- RM este un algoritm offline de planificare a unor taskuri care respecta urmatoarele constrangeri
  - (1) cererile taskurilor cu deadline hard sunt periodice, cu interval constant intre cereri
  - (2) deadline-urile constau exclusiv in constrangeri de executabilitate, i.e., fiecare task trebuie sa termine inainte sa apara urmatoarea cerere pt task
  - (3) taskurile sunt independente, cererile pt un anumit task de initierea sau terminarea cererilor pt alte taskuri
  - (4) timpul de executie al fiecarui task este constant (nu variaza in timp); timpul de executie aici inseamna timpul CPU necesar sa execute taskul fara intrerupere

Rate Monotonic (RM)

- algoritm pt taskuri periodice (interesante pt modelarea usoara si folosirea pe scara larga in industrie/automatizari)
- RM este un algoritm offline de planificare a unor taskuri care respecta urmatoarele constrangeri
  - (5) orice task neperiodic din sistem este special: e fie rutina de initializare fie de recuperare din eroare
    - Inlocuiesc taskurile periodice cand ruleaza dar nu deadline-uri hard, critice

Algoritmul RM

- taskurile au asignate prioritati statice a.i. taskurile cu rate de sosiri mari au prioritate mare
- taskurile se executa preemptiv
- planificatorul alege intotdeauna taskul cu prioritate maxima
- utilizarea *U* a procesorului de catre *n* taskuri

$$U = \sum_{i=1}^{n} \frac{C_i}{Ti}$$

- **Lema**: daca U < ln 2, planificarea RM e fezabila (toate deadline-urile sunt respectate)
- in practica, U < ln 2 e o cerinta conservatoire, exista seturi de taskuri care se pot planifica RM si a caror utilizare depaseste ln 2

Exemplu RM

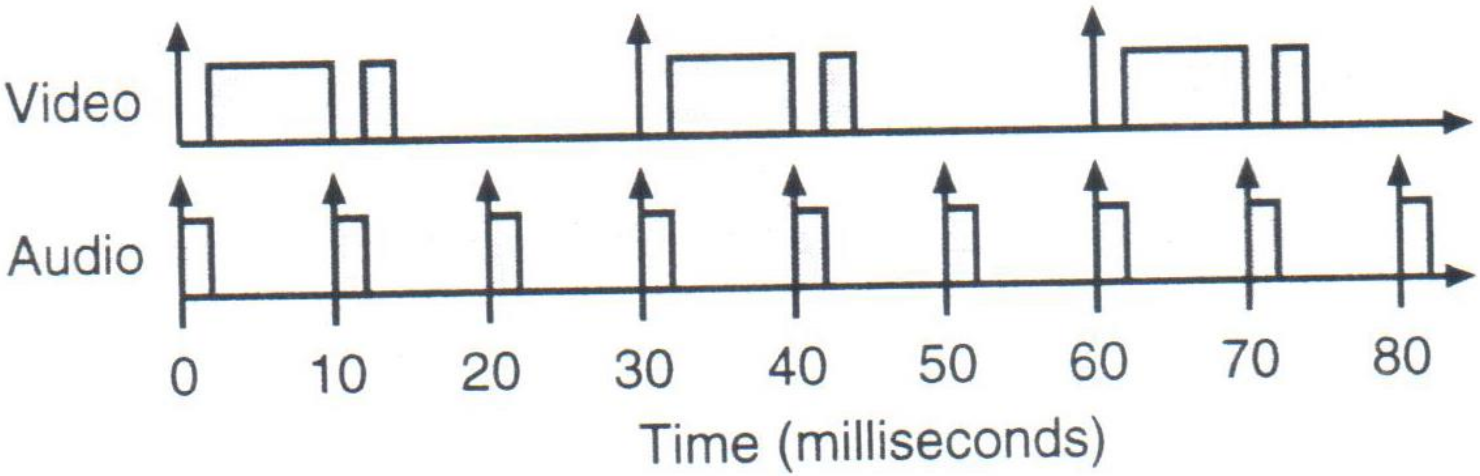
- sistem multimedia
  - task audio cu perioada T_a = 10 ms si C_a = 2 ms
  - task video cu perioada T_v = 30 ms si C_v = 10 ms

  => $U_a = 2/10 = 0.2$

  $U_v = 10/30 = 0.33$

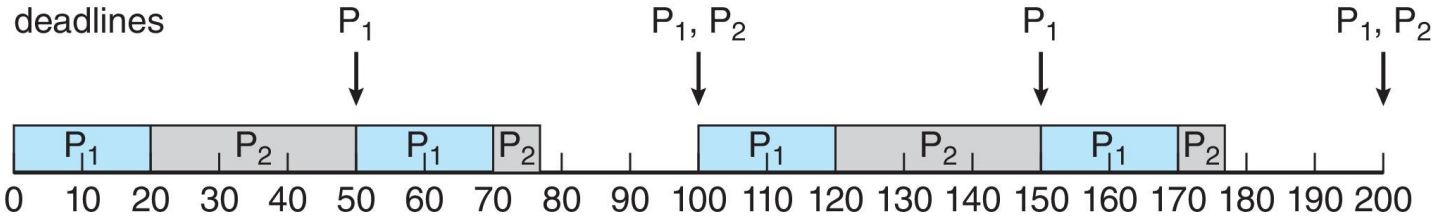
  => U = 0.53 < ln 2 = 0.69 => taskurile sunt planificabile
- taskul audio are frecventa maxima (perioada cea mai scurta) => are si prioritate maxima
- pp. C_v = 20 ms => $U_v = 20/30 = 0.66$=> U = 0.86 > ln 2 ; Totusi, o diagrama de timp arata ca taskurile sunt planificabile RM daca taskul audio are prioritate maxima

Exemplu RM, diagrama de timp



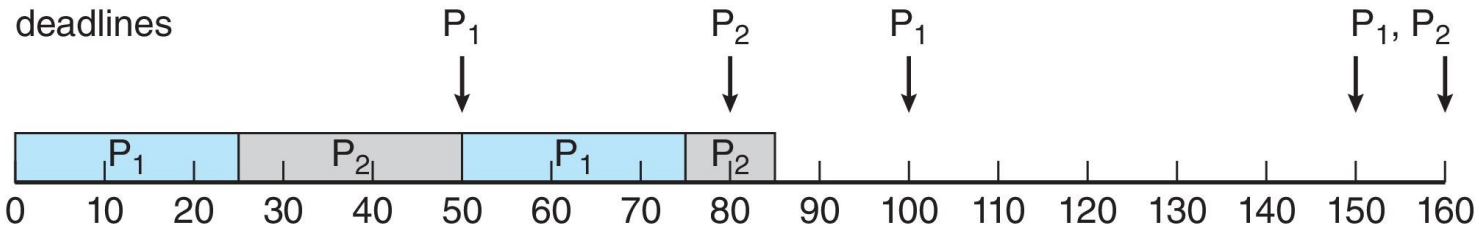
Rate Monotonic Scheduling

- A priority is assigned based on the inverse of its period
- Shorter periods = higher priority;
- Longer periods = lower priority
- P_1 is assigned a higher priority than P_2.



Missed Deadlines with Rate Monotonic Scheduling

- Process P_2 misses finishing its deadline at time 80
- Figure



Earliest Deadline First (EDF)

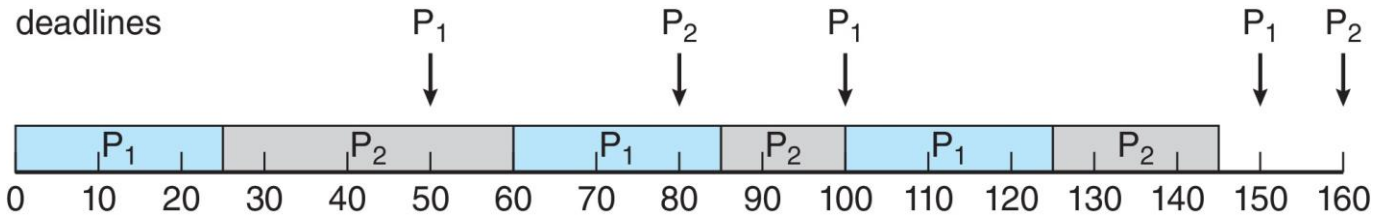
- taskul cu deadline-ul cel mai apropiat are prioritate maxima
- un set de *n* taskuri e planificabil daca

$$U = \sum_{i=1}^{n} \frac{C_i}{Ti} <= 1$$

- ambele exemple de la RM sunt planificabile si EDF

Earliest Deadline First Scheduling (EDF)

- Priorities are assigned according to deadlines:
  - The earlier the deadline, the higher the priority
  - The later the deadline, the lower the priority
- Figure



Least Slack First (LSF)

- *slack* = durata maxima de timp cu care un task poate fi intarziat fara a-si pierde deadline-ul
- LSF alege pt rulare taskul cu slack-ul cel mai mic

Planificarea taskurilor aperiodice

- EDF si LSF se pot folosi si pt. planificarea taskurilor aperiodice
- se poate demonstra ca EDF este optimal
  - daca orice alt algoritm poate produce o planificare fezabila pt un set de taskuri ale caror timpi de sosire, timpi de calcul si deadline-uri sunt cunoscute, atunci si EDF o poate face
- exista o demonstratie similara de optimalitate si pt LSF

Proportional Share Scheduling

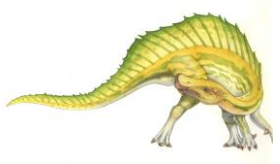
- *T* shares are allocated among all processes in the system
- An application receives *N* shares where *N* < *T*
- This ensures each application will receive ***N*** / *T* of the total processor time

Alocare proportionala de resurse

- planificarea de timp real functioneaza bine in sisteme embedded (mix-ul de job-uri e cunoscut a priori)
- in medii dinamice sistemul poate deveni supraincarcat iar aplicatiile de timp real nu-si pot respecta deadline-urile
- solutiile de planificare bazate pe prioritati nu sunt intotdeauna utile
  - ex: task computational intensiv (video) de prioritate mare intarzie indefinit taskuri non real-time de prioritate mica (compilari)
- abordare noua in planificarea de procese
  - renuntam la conditia ca toate procesele sa termine cu siguranta inainte de deadline
  - planificatorul controleaza rata de executie a proceselor variind “proportia” de resurse (timp CPU) pe care procesele o primesc

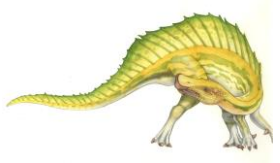
Planificare de tip loterie

- Lottery Scheduling, Waldspurger & Weihl 1994
- un algoritm randomizat de alocare proportionala a resurselor in general
  - timp CPU
  - largime de banda I/O
  - alocare de memorie
  - accesul la lock-uri
- controleaza ratele relative de executie ale proceselor variind procentul de alocare al resursei (eg, timpul CPU pe care-l primeste procesul)
- rata de “consum” a resursei de catre un proces activ este proportionala cu procentul de alocare a resursei respective detinut de proces



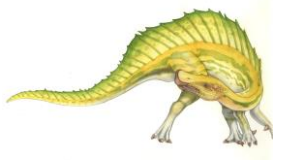
Tichete de loterie

- drepturile asupra resurselor sunt exprimate prin intermediul unor *tichete de loterie*
- alocarea resursei se face prin organizarea unei loterii:
  - resursa e acordata procesului care detine tichetul castigator
  - alocarea e proportionala cu nr de tichete detinut de proces
- tichetele de loterie
  - reprezinta dreptul de a folosi o fractiune din resursa (suma tichetelor detinute de un proces defineste procentul total de utilizare a resursei la care are dreptul procesul)
  - *abstracte*
  - *relative*
  - *uniforme*



Tichete de loterie (cont'd)

- tichetele de loterie
  - *abstracte*: cuantifica dreptul de folosire a resursei independent de detaliile HW
  - *relative*: fractiunea de resursa pe care o cuantifica variaza proportional cu nr total de tichete detinute
    - variatia e dinamica, in functie de competitia la resursa
    - un proces va obtine o fractiune mai mare a unei resurse cu competitie mica decat in cazul uneia cu competie mare
    - in cel mai rau caz, un proces primeste o fractiune de resursa proportionala cu nr sau de tichete
  - *uniforme*: indiferent de tipul de resursa, procentele de alocare ale proceselor sunt reprezentate omogen prin tichete



Algoritm

- la inceputul fiecarei cuante, planificatorul organizeaza o loterie si resursa (CPU) e acordata detinatorului tichetului castigator
- tichetul castigator este generat aleator cf unei distributii uniforme
- apoi se cauta in lista de procese gata de rulare participantul care detine tichetul castigator
  - acesta este participantul pentru care suma cumulate de tichete este mai mare decat tichetul castigator
- Ex: nr total de tichete = 20, 5 procese cu [10, 2, 5, 1, 2] tichete

  random[0..19] = 15

  S1 = 10; S1 > 15? NU

  S2 = 12; S2 > 15? NU

  S3 = 17; S3 > 15? DA => procesul cu nr. 3 (detine 5 tichete) primeste procesorul

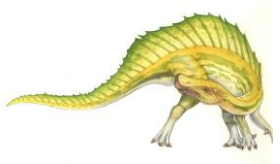
Analiza algoritmului

- running time
  - generarea de nr aleatoare e rapida (cca 10 instr RISC)
  - traversarea listei de procese O(n)
  - acumularea sumei partiale
- optimizari pt reducere nr de procese ce trebuie examinate
  - (1) pt distributie inegala de tichete, ordoneaza descrescator lista de procese cf nr de tichete detinute

    => lungimea medie de cautare scade pt ca procesele cu nr mare de tichete sunt frecvent selectate
  - pt *n* mare, se poate folosi un arbore cu sume partiale in nodurile interne si nr de tichete ale proceselor pe frunze

    => localizarea tichetului castigator se face pornind din radacina

    catre procesul (frunza) castigator => complexitate O(log n)



Caracteristici LS

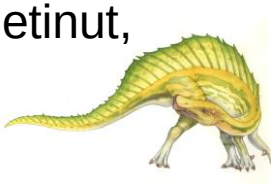
- (1) *fair* dpdv probabilistic
  - dupa o perioada suficienta de timp, valoarea estimata a fractiunii de alocare a resursei este proportionala cu nr de tichete detinut de proces
  - obs: nu e garantat ca un proces cu *t* tichete din totalul *T* va primi **exact** *t / T* din utilizarea resursei !
  - diferenta scade insa pe masura ce nr de alocari creste
  - Explicatie: fie X = nr de loterii castigate de un proces, variabila aleatoare cu distributie binomiala

  Fie $p = t / T$ probabilitatea ca un proces cu *t* tichete sa castige loteria

  - dupa *n* loterii identice $E[X] = np$ iar $Var(x) = np(1 – p)$

  $\Rightarrow$ coef. de variatie = $\frac{stdev(X)}{E[X]} = \sqrt{\frac{1-p}{np}}$

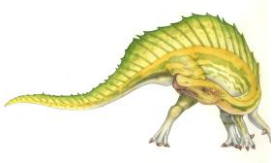
  Adica, throughput-ul procesului e proportional cu nr de tichete detinut, iar acuratetea estimarii se imbunatateste cu viteza lui $\sqrt{n}$



Caracteristici LS (cont'd)

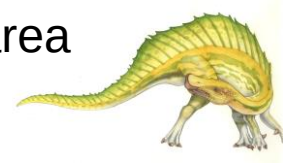
- (1) *fair* dpdv probabilistic
  - fie Y = nr de loterii necesare ca un proces sa castige prima data loteria, variabila aleatoare cu distributie geometrica
  - *E[Y]* = $\frac{1}{p}$ iar *Var(x)* = $\frac{1-p}{p^2}$

  $\Rightarrow$ timpul mediu de raspuns este invers proportional cu nr de tichete detinut de proces



Caracteristici LS (cont'd)

- (2) tichetele se pot transfera de la un proces la altul atunci cand acesta se blocheaza dintr-un motiv oarecare
  - ex: proces client asteapta raspuns de la server, clientul poate transfera temporar tichetele sale serverului
  - clientul poate transfera tichetele sale catre mai multe servere de la care asteapta un raspuns
  - solutie pt. priority inversion
- (3) *inflatia* de tichete: alternativa la transferul de tichete
  - procesul isi mareste dreptul la resurse prin creearea artificiala de tichete pt sine
  - metoda se poate folosi intre procese care au incredere reciproca intre ele
  - prin inflatie/deflatie de tichete procesele controleaza alocarea resursei fara sa comunice explicit intre ele
  - in general este de evitat, pt ca poate conduce la monopolizarea resursei



Caracteristici LS (cont'd)

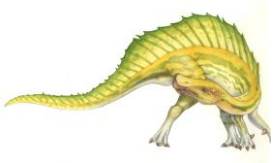
- (4) *tichete de compensare*
  - cand un proces consuma doar o fractiune *f* din cuanta alocata pentru utilizarea resursei, el primeste un *tichet de compensare*
  - tichetul de compensare ii creste procesului valoarea cu *1 / f* pana cand procesul primeste urmatoarea cuanta
  - astfel, consumul resursei de *f * p* este ajustat cu *1 / f* ca sa corespunda fractiunii reprezentate de p
  - in absenta unei astfel de metode, procesul care nu consuma intreaga fractiune alocata utilizeaza resursa mai putin decat are dreptul

POSIX Real-Time Scheduling

- The POSIX.1b standard
- API provides functions for managing real-time threads
- Defines two scheduling classes for real-time threads:
  1. SCHED_FIFO - threads are scheduled using a FCFS strategy with a FIFO queue. There is no time-slicing for threads of equal priority
  2. SCHED_RR - similar to SCHED_FIFO except time-slicing occurs for threads of equal priority
- Defines two functions for getting and setting scheduling policy:
  1. `pthread_attr_getsched_policy(pthread_attr_t *attr, int *policy)`
  2. `pthread_attr_setsched_policy(pthread_attr_t *attr, int policy)`

POSIX Real-Time Scheduling API

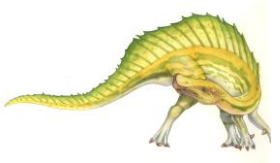
```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5
int main(int argc, char *argv[])
{
   int i, policy;
   pthread_t tid[NUM_THREADS];
   pthread_attr_t attr;
   /* get the default attributes */
   pthread_attr_init(&attr);
   /* get the current scheduling policy */
   if (pthread_attr_getschedpolicy(&attr, &policy) != 0)
      fprintf(stderr, "Unable to get policy.\n");
   else {
      if (policy == SCHED_OTHER) printf("SCHED_OTHER\n");
      else if (policy == SCHED_RR) printf("SCHED_RR\n");
      else if (policy == SCHED_FIFO) printf("SCHED_FIFO\n");
   }
```



POSIX Real-Time Scheduling API (Cont.)

```
   /* set the scheduling policy - FIFO, RR, or OTHER */
   if (pthread_attr_setschedpolicy(&attr, SCHED_FIFO) != 0)
      fprintf(stderr, "Unable to set policy.\n");
   /* create the threads */
   for (i = 0; i < NUM_THREADS; i++)
      pthread_create(&tid[i],&attr,runner,NULL);
   /* now join on each thread */
   for (i = 0; i < NUM_THREADS; i++)
      pthread_join(tid[i], NULL);
}

/* Each thread will begin control in this function */
void *runner(void *param)
{
   /* do some work ... */
   pthread_exit(0);
}
```



Operating System Examples

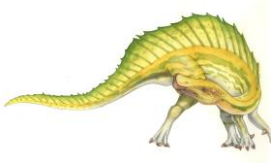
- Linux scheduling
- Windows scheduling
- Solaris scheduling

Linux Scheduling Through Version 2.5

- Prior to kernel version 2.5, ran variation of standard UNIX scheduling algorithm
- Version 2.5 moved to constant order *O*(1) scheduling time
  - Preemptive, priority based
  - Two priority ranges: time-sharing and real-time
  - **Real-time** range from 0 to 99 and **nice** value from 100 to 140
  - Map into global priority with numerically lower values indicating higher priority
  - Higher priority gets larger q
  - Task run-able as long as time left in time slice (**active**)
  - If no time left (**expired**), not run-able until all other tasks use their slices
  - All run-able tasks tracked in per-CPU **runqueue** data structure
    - Two priority arrays (active, expired)
    - Tasks indexed by priority
    - When no more active, arrays are exchanged
  - Worked well, but poor response times for interactive processes

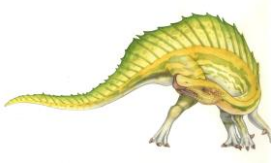
Linux Scheduling in Version 2.6.23 +

- **Completely Fair Scheduler (CFS)**
- **Scheduling classes**
  - Each has specific priority
  - Scheduler picks highest priority task in highest scheduling class
  - Rather than quantum based on fixed time allotments, based on proportion of CPU time
  - Two scheduling classes included, others can be added
    1. default
    2. real-time



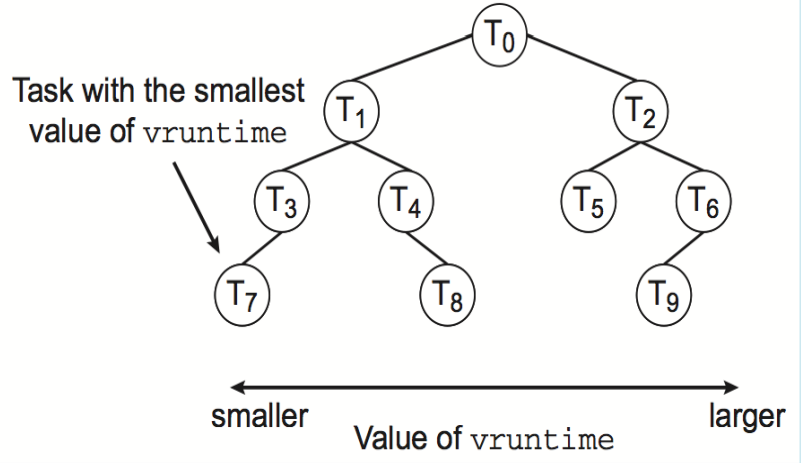
Linux Scheduling in Version 2.6.23 + (Cont.)

- Quantum calculated based on **nice value** from -20 to +19
  - Lower value is higher priority
  - Calculates **target latency** – interval of time during which task should run at least once
  - Target latency can increase if say number of active tasks increases
- CFS scheduler maintains per task **virtual run time** in variable **`vruntime`**
  - Associated with decay factor based on priority of task – lower priority is higher decay rate
  - Normal default priority yields virtual run time = actual run time
- To decide next task to run, scheduler picks task with lowest virtual run time



CFS Performance

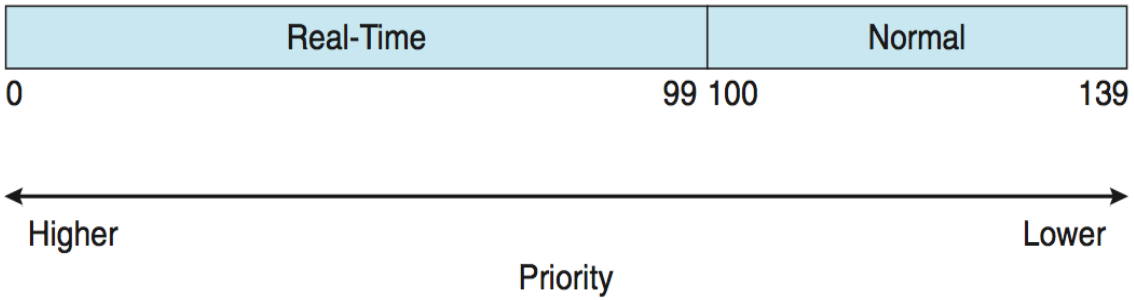
The Linux CFS scheduler provides an efficient algorithm for selecting which task to run next. Each runnable task is placed in a red-black tree—a balanced binary search tree whose key is based on the value of `vruntime`. This tree is shown below:



When a task becomes runnable, it is added to the tree. If a task on the tree is not runnable (for example, if it is blocked while waiting for I/O), it is removed. Generally speaking, tasks that have been given less processing time (smaller values of `vruntime`) are toward the left side of the tree, and tasks that have been given more processing time are on the right side. According to the properties of a binary search tree, the leftmost node has the smallest key value, which for the sake of the CFS scheduler means that it is the task with the highest priority. Because the red-black tree is balanced, navigating it to discover the leftmost node will require $O(\lg N)$ operations (where N is the number of nodes in the tree). However, for efficiency reasons, the Linux scheduler caches this value in the variable `rb_leftmost`, and thus determining which task to run next requires only retrieving the cached value.

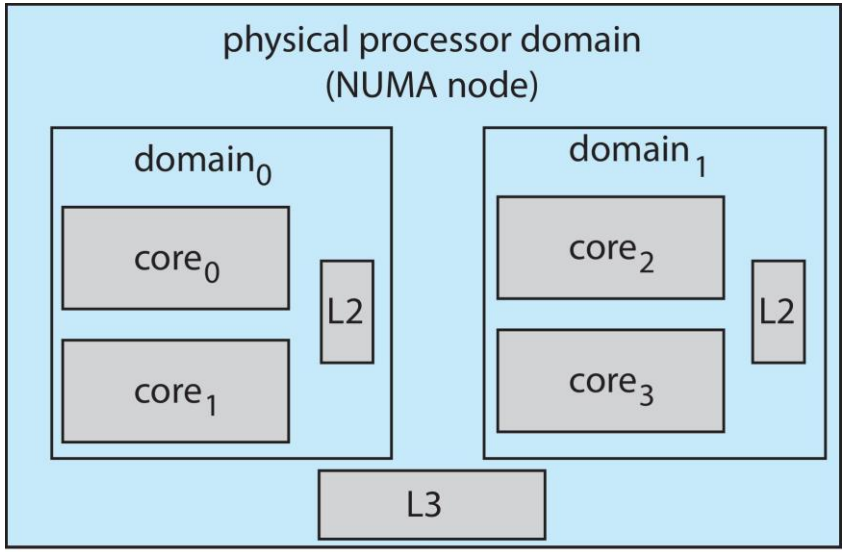
Linux Scheduling (Cont.)

- Real-time scheduling according to POSIX.1b
  - Real-time tasks have static priorities
- Real-time plus normal map into global priority scheme
- Nice value of -20 maps to global priority 100
- Nice value of +19 maps to priority 139



Linux Scheduling (Cont.)

- Linux supports load balancing, but is also NUMA-aware.
- **Scheduling domain** is a set of CPU cores that can be balanced against one another.
- Domains are organized by what they share (i.e., cache memory.) Goal is to keep threads from migrating between domains.



Windows Scheduling

- Windows uses priority-based preemptive scheduling
- Highest-priority thread runs next
- **Dispatcher** is scheduler
- Thread runs until (1) blocks, (2) uses time slice, (3) preempted by higher-priority thread
- Real-time threads can preempt non-real-time
- 32-level priority scheme
- **Variable class** is 1-15, **real-time class** is 16-31
- Priority 0 is memory-management thread
- Queue for each priority
- If no run-able thread, runs **idle thread**

Windows Priority Classes

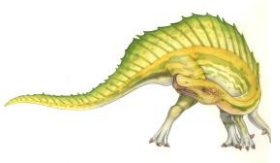
- Win32 API identifies several priority classes to which a process can belong
  - REALTIME_PRIORITY_CLASS, HIGH_PRIORITY_CLASS, ABOVE_NORMAL_PRIORITY_CLASS,NORMAL_PRIORITY_CLASS, BELOW_NORMAL_PRIORITY_CLASS, IDLE_PRIORITY_CLASS
  - All are variable except REALTIME
- A thread within a given priority class has a relative priority
  - TIME_CRITICAL, HIGHEST, ABOVE_NORMAL, NORMAL, BELOW_NORMAL, LOWEST, IDLE
- Priority class and relative priority combine to give numeric priority
- Base priority is NORMAL within the class
- If quantum expires, priority lowered, but never below base

Windows Priority Classes (Cont.)

- If wait occurs, priority boosted depending on what was waited for
- Foreground window given 3x priority boost
- Windows 7 added **user-mode scheduling** (**UMS**)
  - Applications create and manage threads independent of kernel
  - For large number of threads, much more efficient
  - UMS schedulers come from programming language libraries like C++ **Concurrent Runtime** (ConcRT) framework

Windows Priorities

	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



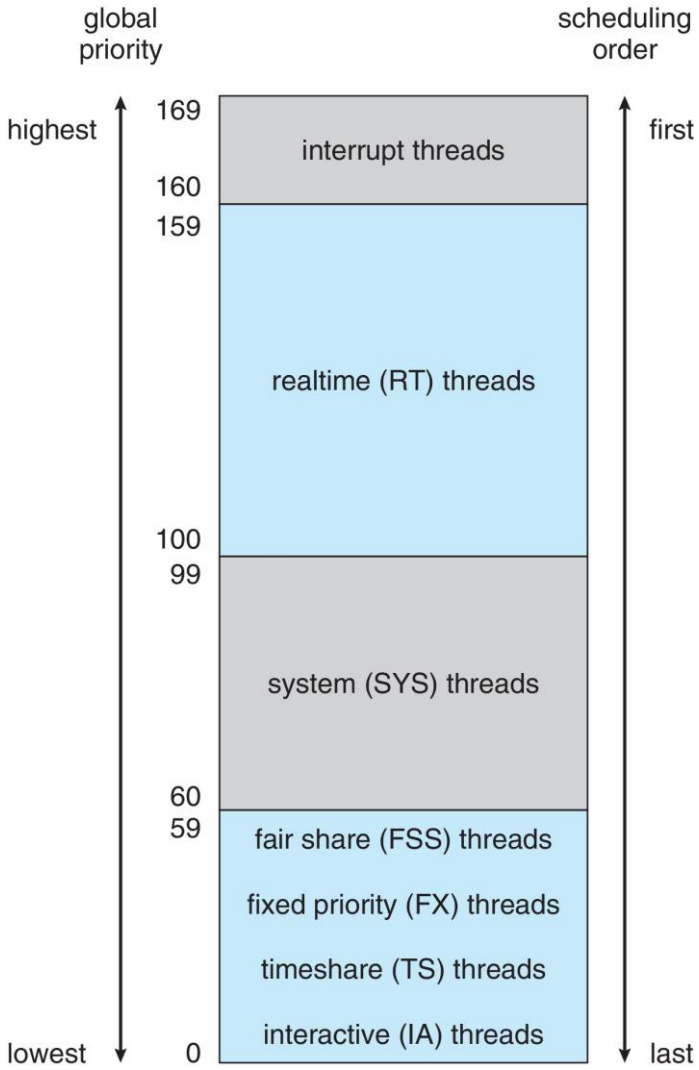
Solaris

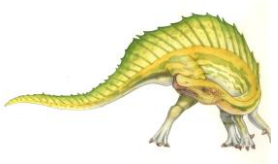
- Priority-based scheduling
- Six classes available
  - Time sharing (default) (TS)
  - Interactive (IA)
  - Real time (RT)
  - System (SYS)
  - Fair Share (FSS)
  - Fixed priority (FP)
- Given thread can be in one class at a time
- Each class has its own scheduling algorithm
- Time sharing is multi-level feedback queue
  - Loadable table configurable by sysadmin

Solaris Dispatch Table

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

Solaris Scheduling





Solaris Scheduling (Cont.)

- Scheduler converts class-specific priorities into a per-thread global priority
  - Thread with highest priority runs next
  - Runs until (1) blocks, (2) uses time slice, (3) preempted by higher-priority thread
  - Multiple threads at same priority selected via RR

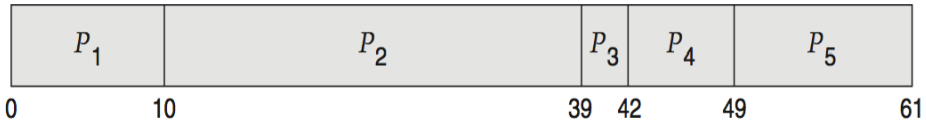
Algorithm Evaluation

- 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:

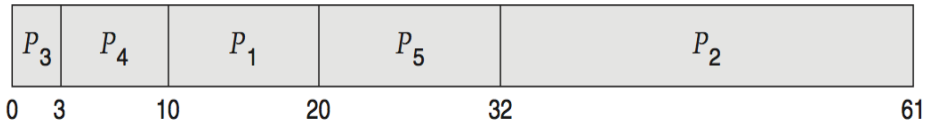
Process	Burst Time
P_1	10
P_2	29
P_3	3
P_4	7
P_5	12

Deterministic Evaluation

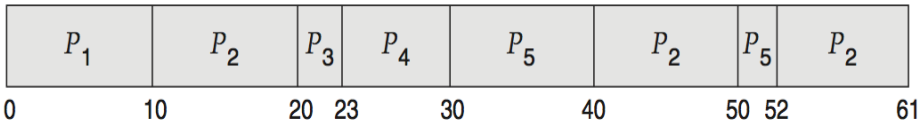
- For each algorithm, calculate minimum average waiting time
- Simple and fast, but requires exact numbers for input, applies only to those inputs
  - FCS is 28ms:



  - Non-preemptive SFJ is 13ms:

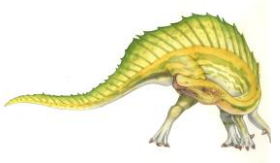


  - RR is 23ms:



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.

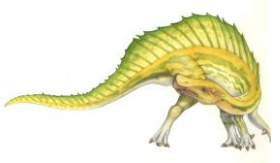


Little's Formula

- 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:

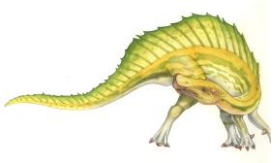
  $n = \lambda \times W$

  - 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

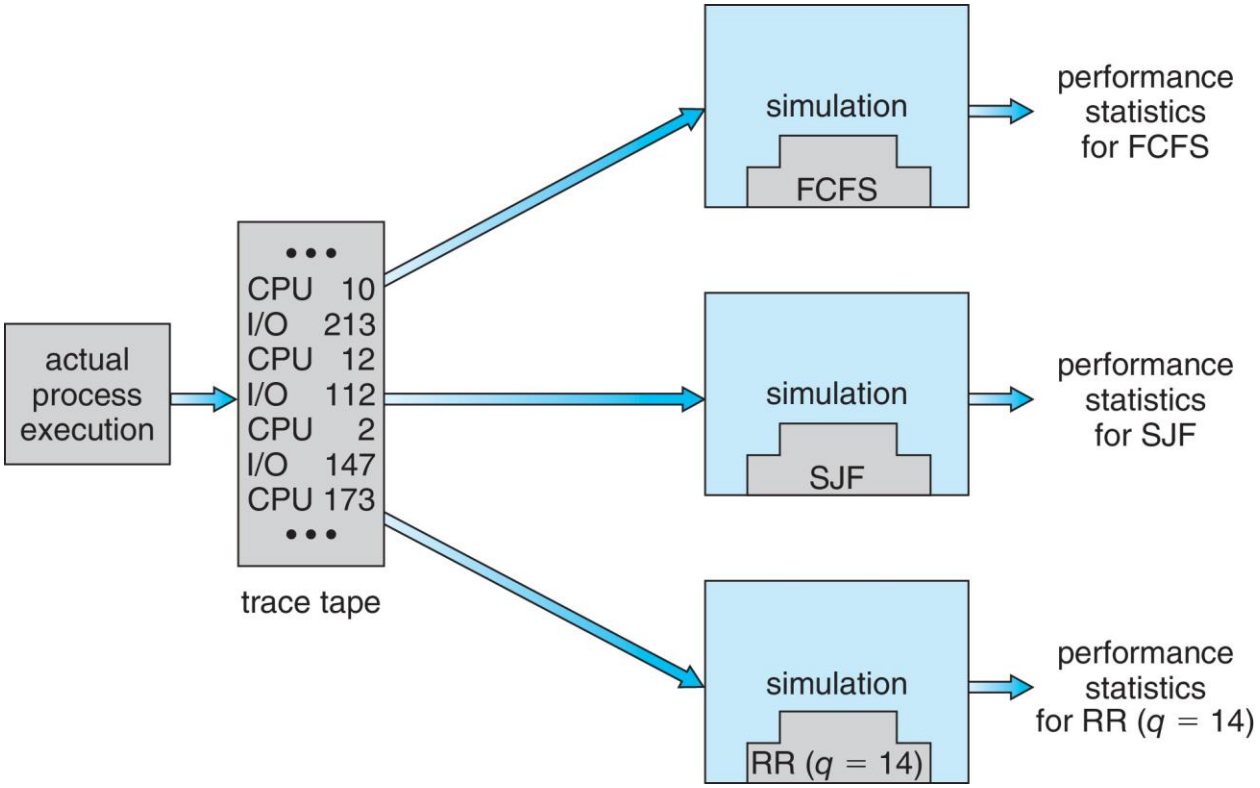


Simulations

- 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

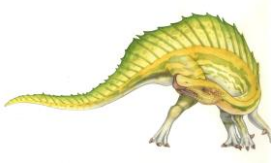


Evaluation of CPU Schedulers by Simulation



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



End of Chapter 5