PMR5230
Sistemas Computacionais para Automação
Aula 03

Fabio Kawaoka Takase, Newton Maruyama

Embedded Systems Laboratory

1 de Outubro de 2008

Referências Bibliográficas

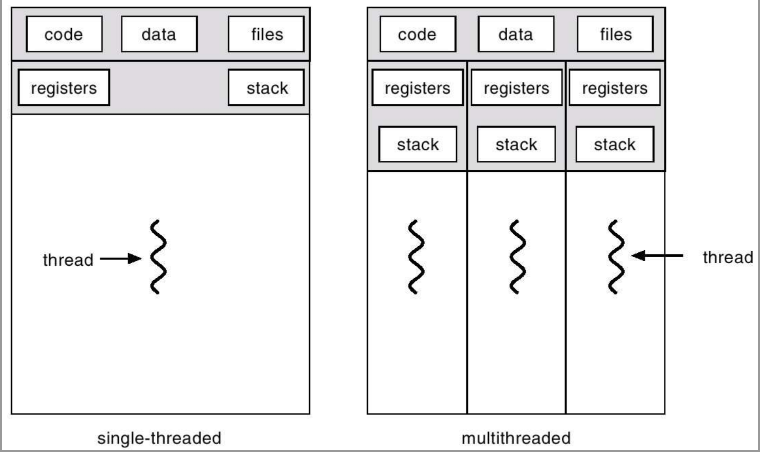
Livro do Silberschatz *Operating System Concepts*

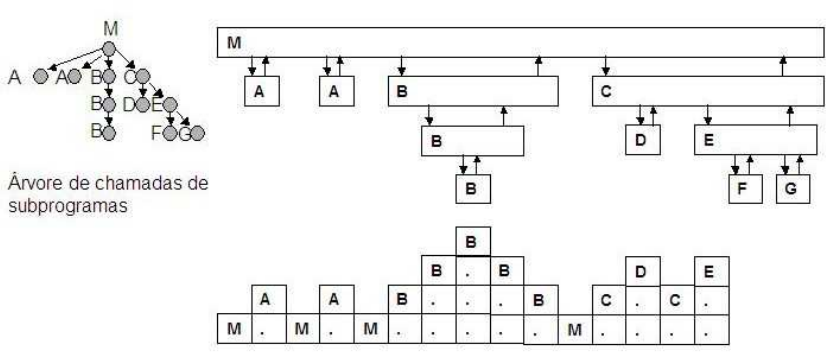
- Capítulo 5
- Capítulo 6

Threads

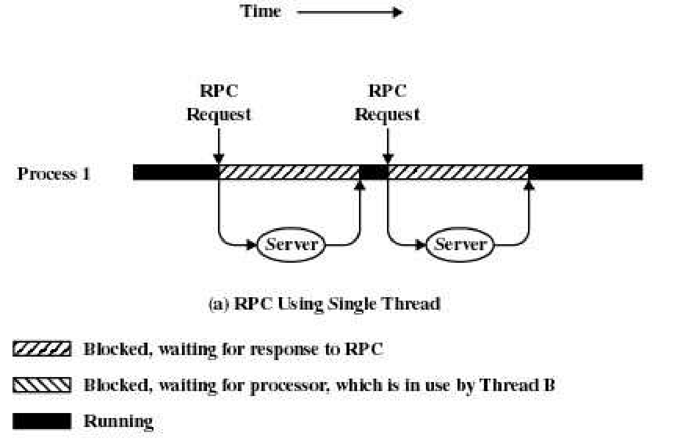
- A thread is a lightweight process (LWP)
- It comprises:
  - A program counter,
  - A register set,
  - A stack.
- It shares with other threads belonging to the same process:
  - Its code section,
  - Data section,
  - Open files, etc.
- A traditional process (heavyweight) has a single thread of control.

Single and Multithreaded Processes

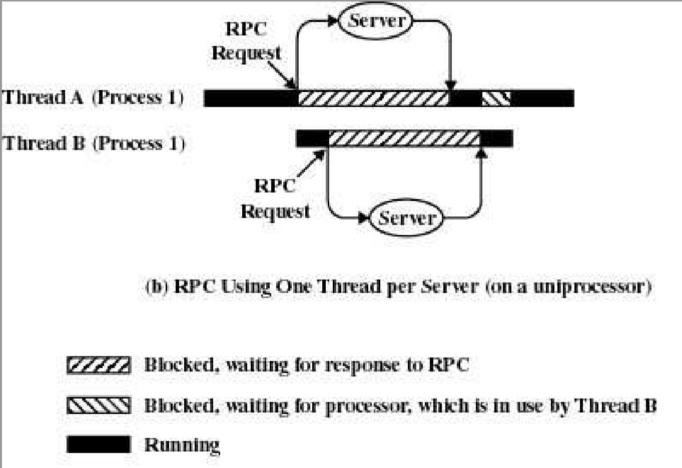



M
A
A
B
C
B
D
E
B
F
G
Árvore de chamadas de subprogramas
M
A
A
B
C
B
D
E
B
F
G
B
B
B
D
E
A
A
B
B
C
C
M
M
M
M

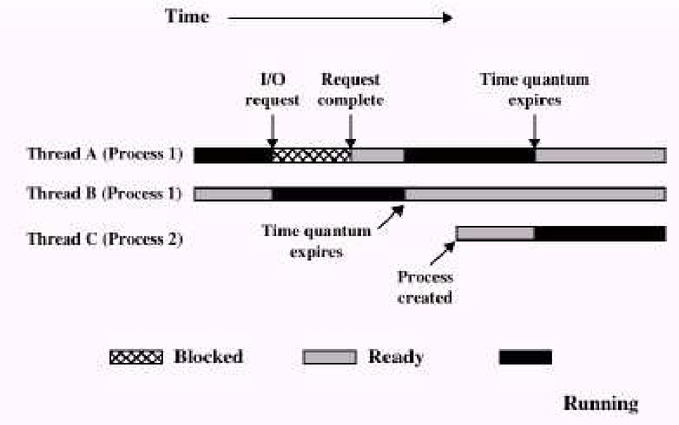
Remote Procedure Call Using Threads - 1



Remote Procedure Call Using Threads - 2



Remote Procedure Call Using Threads - 3



Benefícios da Utilização de Threads

Responsiveness allow a program to continue running even if part of it is blocked or performing a lengthy operation.

Resource Sharing code, data and files are shared.

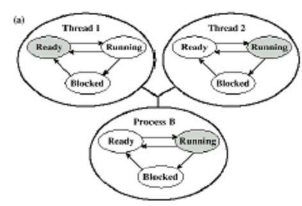
Economy allocation of memory and other resources are costly. In Solaris 2 creating a process is 30 times slower than is creating a thread, and context switching is about five times slower.

Utilization of MP Architectures a single threaded process can only run in one CPU. Multithreading allow each thread to run on a different processor.

User Threads

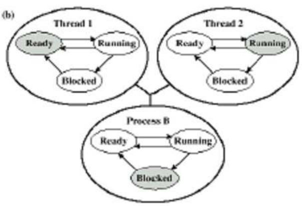
- Thread management done by user-level threads library
- Fast to create and manage.
- Examples
  - POSIX Pthreads
  - Mach C-threads
  - Solaris threads

Relationship between user level threads states and process states



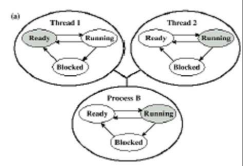
Process P_B is executing in its thread t_2. The application executing in t_2 makes a system call that blocks B. For example, an IO call is made. This causes control to transfer to the kernel. The kernel invokes the IO action, places P_B in the blocked state and switches to another process.

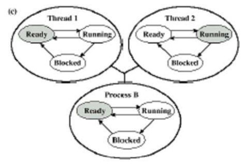
Relationship between user level threads states and process states



Meanwhile, according to the data structure maintained by the threads library, t_2 of P_B is still in the running state. It is important to note that t_2 is not actually running in the sense of being executed, but it is perceived as being in the running state by the threads library.

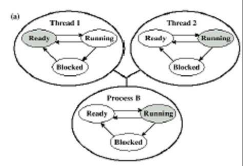
Relationship between user level threads states and process states

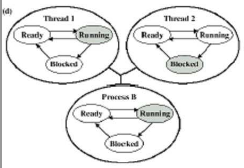




Process P_B is executing in its thread t_2. A clock interrupt passes control to the kernel and the kernel determines that the currently running process P_B has exhausted its time slice. The kernel places P_B in the ready state and switches to another process. Meanwhile, t_2 of P_B is still in the running state.

Relationship between user level threads states and process states





Process P_B is executing in its thread t_2. t_2 has reached a point where it needs some action performed by thread 1 of process B. t_2 enters a blocked state and thread 1 transitions from ready to running. The process itself remains in the running state.

Kernel Threads

- Supported by the Kernel
- Examples
  - Windows 95/98/NT/2000
  - Solaris
  - Tru64 UNIX
  - BeOS
  - Linux

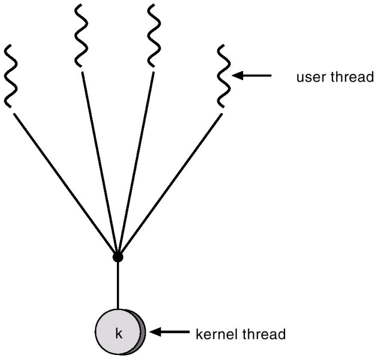
Multithreading Models

- Many-to-One
- One-to-One
- Many-to-Many

Many-to-One

- Many user-level threads mapped to single kernel thread.
- Only one thread can access the kernel at a time.
- Entire process is blocked if a thread makes a blocking system call
- Used on systems that do not support kernel threads.

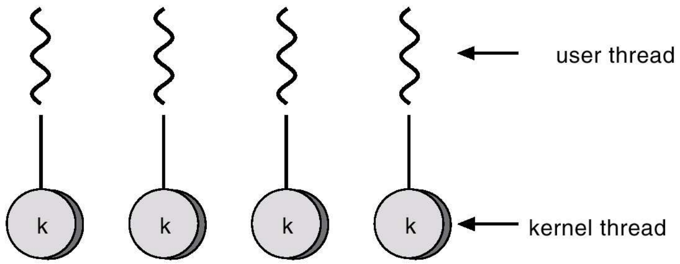
Many-to-One



One-to-One

- Each user-level thread maps to kernel thread.
- Provides more concurrency than the many-to-one model by allowing another thread to run when a thread makes a system call.
- Allows multiple threads to run in parallel on multiprocessors.
- The drawback of this model is that creating a user thread requires creating the corresponding kernel thread and this creates an important overhead.
- Examples
  - Windows 95/98/NT/2000
  - OS/2

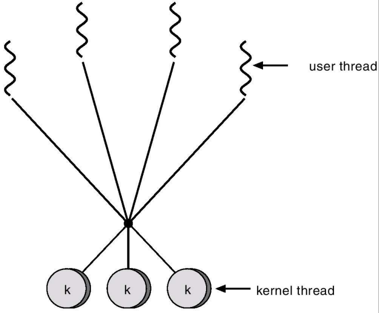
One-to-One



Many-to-Many

- Allows many user level threads to be mapped to many kernel threads.
- Developers can create as many user threads as necessary and the corresponding kernel thread can run in parallel on a multiprocessor.
- When a thread performs a blocking system call, the kernel can schedule another thread execution.
- Solaris 2, IRIX, HP-UX

Many-to-Many



Threading Issues

- Semantics of fork() and exec() system calls.
- Thread cancellation.
- Signal handling
- Thread pools
- Thread specific data

The Fork and Exec System calls

- In a multithreaded program the semantics of the fork and exec system calls change.
- If one thread in a program calls fork, does the new process duplicate all threads or is the new process single threaded ?
- Some UNIX systems have chosen to have two versions of fork, one that duplicates all threads and another that duplicates only the thread that invoked the fork system call.

The Fork and Exec System calls

- If exec is called immediately after forking then duplicating all threads are unnecessary as the program specified will replace the entire process. Duplicating only the calling thread is appropriate.
- If the separate process does not call exec after forking the separate process will duplicate all threads.

Thread Cancellation

- Thread cancellation is the task of terminating a thread before it is completed.
- For exemple, if multiple threads are concurrently searching through a database and one thread returns the result, the remaining threads might be cancelled.
- Another situation might occur when a user presses a button on a web browser that stops a web page from loading any further. Often a web page is loaded in a separate thread.
- A thread that is to be cancelled is called target thread.

Thread Cancellation

- Cancellation may occur in two different scenarios:
  - Asynchronous cancellation one thread immediately terminates the target thread.
  - Deferred cancellation the target thread can periodically check if it should terminate, allowing the target thread to terminate in an orderly fashion.
- The difficulty with cancellation arise in situations where resources have been allocated to a cancelled thread or if a thread was cancelled while in the middle of updating data it is sharing with other threads.
- Specially during an asynchronous cancellation a necessary system wide resource might not be released.

Signal Handling

- A signal is used to notify a process that a particular event has occured.
- A signal may be received either synchronously or asynchronously.
- All signals follow the same pattern:
    1. A signal is generated by the occurence of a particular event.
    2. A generated signal is delivered to a process.
    3. Once delivered, the signal must be handled.
- In a single threaded program signals are always delivered to a process.
- In multithreaded programs a process may have several threads, where then should the signal be delivered ?

Signal Handling

- The following options exist:
  - Deliver the signal to the thread to which the signal applies.
  - Deliver the signal to every thread in the process.
  - Deliver the signal to certain threads in the process.
  - Assign a specific thread to receive all signals for the process. (SOLARIS 2)
- Windows 2000 does not support explicitly for signals, but it can be emulated using Asynchronous Procedure Calls (APC).

Thread Pools

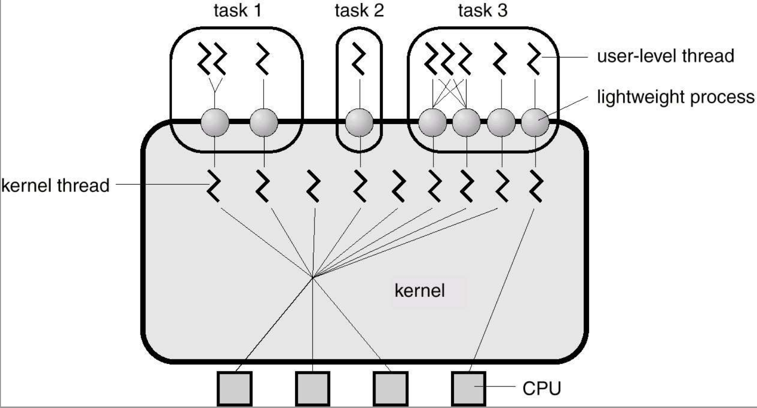
- If we allow all concurrent requests to be serviced in a new thread systems resources can be exhausted.
- One possible solution is the creation of thread pools.
- The idea behind a thread pools is to create a number of threads at process startup and place them into a pool where they sit and wait for work.
- When a server receives a request, it awakes a thread from this pool passing it the request to service.
- Once the thread completes its service it returns to the pool.
- If the pool contains no available thread, the server waits until one becomes free.

Thread-Specific Data

- Threads belonging to a process share the data of the process.
- However, each thread might need its own copy of certain data in some circumstances, this is called ***thread specific data***.
- For example, in a transaction processing system we might service each transaction in a separate thread.

Pthreads

- a POSIX standard (IEEE 1003.1c) *Application Programming Interface* API for thread creation and synchronization.
- API specifies behavior of the thread library, implementation is up to development of the library.
- Common in UNIX operating systems.


task 1
task 2
task 3
user-level thread
lightweight process
kernel thread
kernel
CPU

Windows 2000 Threads

- Implements the one-to-one mapping.
- Each thread contains
  - a thread id
  - register set
  - separate user and kernel stacks
  - private data storage area

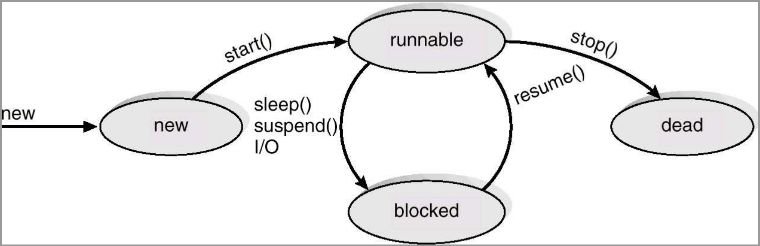
Linux Threads

- Linux refers to them as *tasks* rather than *threads*.
- Thread creation is done through clone() system call.
- clone() allows a child task to share the address space of the parent task (process)

Java Threads

- Java threads may be created by:
  - Extending Thread class
  - Implementing the Runnable interface
- Java threads are managed by the JVM.

Java Thread States



Java Threads

```
class Somatoria extends Thread{
  private int limite;
    public Somatoria(int n){   limite = n;  }
    public void run(){
      int somatoria = 0;
      for (int i=1;i<=limite;i++) somatoria +=i;
      System.out.println("Somatoria = "+ somatoria);
    }
}

public class TesteSomatoria{
  public static void main(String[] args){
    Somatoria threadSomatoria = new Somatoria(10);
    threadSomatoria.start();
  }
}
```

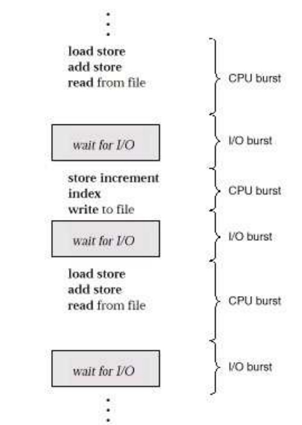
- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Multiple-Processor Scheduling
- Real-Time Scheduling
- Algorithm Evaluation

Basic Concepts

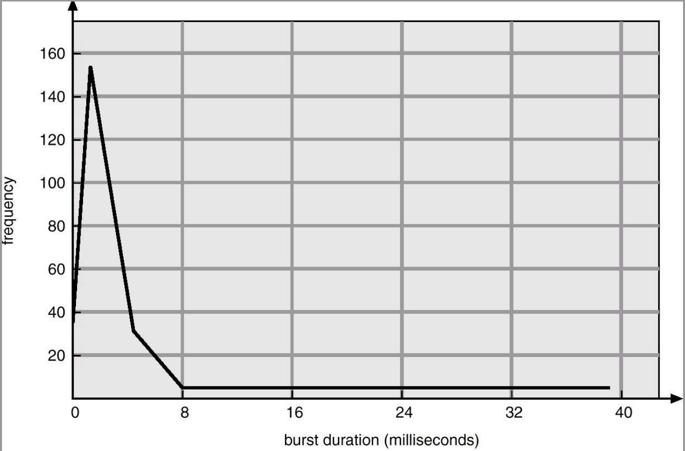
- Maximum CPU utilization obtained with multiprogramming
- CPU-I/O Burst Cycle - Process execution consists of a cycle of CPU execution and I/O wait.
- CPU burst distribution
  - An I/O bound program typically have many very short CPU bursts.
  - A CPU bound program might have a few very long CPU bursts.

"To burst into or out of a place means to enter or leave it suddenly with a lot of energy or force"
(Collins CoBuild)

Alternating Sequence of CPU And I/O Bursts



Histogram of CPU-burst Times



CPU Scheduler

Information associated with each process.

- Selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them.
- CPU scheduling decisions may take place when a process:
  1. Switches from running to waiting state (for example, due to an I/O request).
  2. Switches from running to ready state (For example, when an interrupt occurs).
  3. Switches from waiting to ready (For example, due to a completion of I/O).
  4. Terminates.
- Scheduling under 1 and 4 is *nonpreemptive*.
- All other scheduling is *preemptive*.

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.

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, i.e. time spent waiting to get into memory + waiting in the ready queue, executing on the CPU and doing I/O.

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, **not** output (for time-sharing environment)

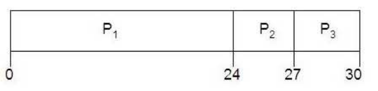
Optimization Criteria

- 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

- Arrival order: P_1 , P_2 , P_3 Gantt Chart:

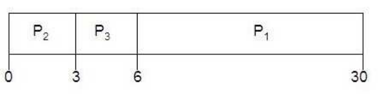


- Waiting time for $P_1 = 0$; $P_2 = 24$; $P_3 = 27$
- Average waiting time: $(0 + 24 + 27)/3 = 17$

First-Come, First-Served (FCFS) Scheduling

Process	Burst Time
P_1	24
P_2	3
P_3	3

- Arrival order: P_2 , P_3 , P_1 Gantt Chart:



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

Shortest-Job-First (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:

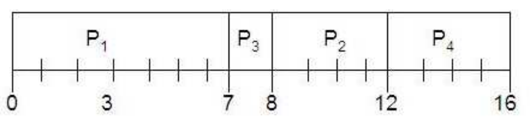
  nonpreemptive once CPU given to the process it cannot be preempted until completes its CPU burst.

  preemptive if a new process arrives with CPU burst length less than remaining time of current executing process, preempt. This scheme is know as the Shortest-Remaining-Time-First (SRTF).
- SJF is optimal - gives minimum average waiting time for a given set of processes.

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)

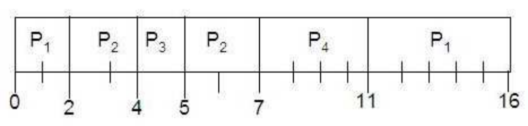


- Average waiting time = (0 + 6 + 3 + 7)/4 = 4

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)



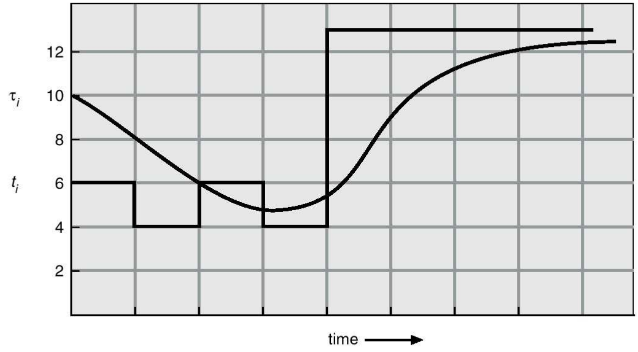
- Average waiting time = (9 + 1 + 0 +2)/4 = 3

Determining Length of Next CPU Burst

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

$\tau_{n+1} = \alpha \cdot t_n + (1 - \alpha) \cdot \tau_n$
where
t_n = actual lenght of n^{th} CPU Burst
τ_{n+1} = predicted value for the next CPU Burst
$0 \leq \alpha \leq 1$

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.
- $\alpha = 1$
  - $\tau_{n+1} = t_n$
  - Only the actual last CPU burst counts.
- If we expand the formula, we get:
  $\tau_{n+1} = \alpha t_n + (1 - \alpha)\alpha t_{n-1} + ...$
  $(1 - \alpha)^j \alpha t_{n-j} + ...$
  $(1 - \alpha)^{n-1} \alpha t_n \tau_0$
- Since both α and $(1 - \alpha)$ are less than or equal to 1, each successive term has less weight than its predecessor.

Priority Scheduling

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

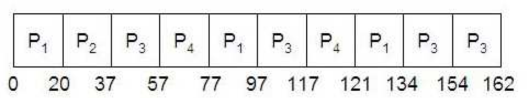
Round Robin (RR)

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

Example of RR with Time Quantum = 20

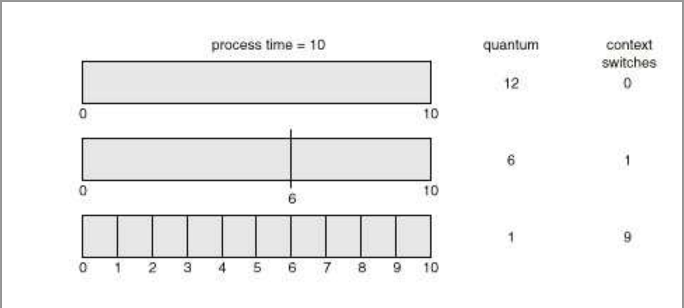
Process	Burst Time
P_1	53
P_2	17
P_3	68
P_4	24

- The Gantt Chart for the schedule is:

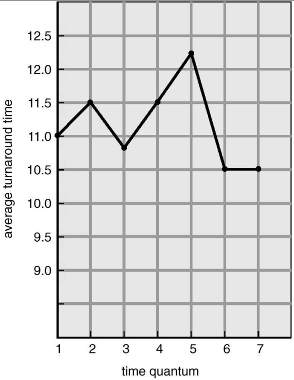


- Typically, higher average turnaround than SJF, but better response.

Time Quantum and Context Switch Time



Turnaround Time Varies With The Time Quantum

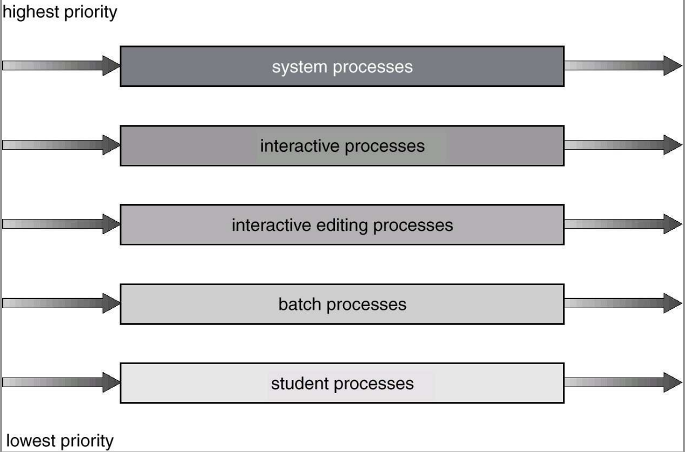


process	time
P_1	6
P_2	3
P_3	1
P_4	7

Multilevel Queue

- Ready queue is partitioned into separate queues:
  foreground (interactive)
  background (batch)
- Each queue has its own scheduling algorithm,
  foreground - RR
  background - FCFS
- Scheduling must be done between the queues.
  - Fixed priority scheduling; (i.e., serve all from foreground then from background). 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 in RR
    - 20% to background in FCFS

Multilevel Queue Scheduling



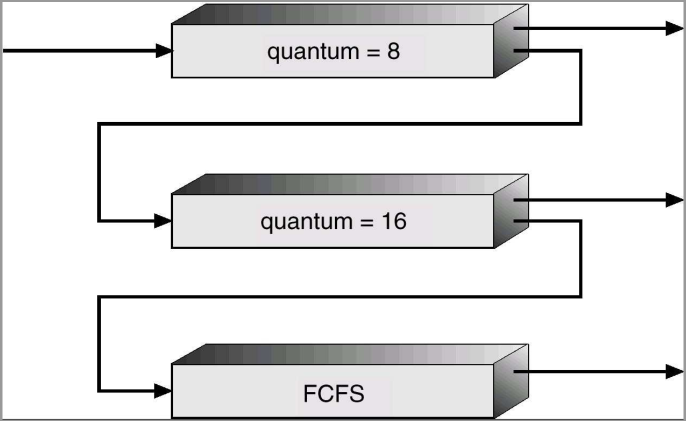
Multilevel Feedback Queue

- A process can move between the various queues; aging can be implemented this way.
- 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

Example of Multilevel Feedback Queue

- Three queues:
  - Q_0 - time quantum 8 milliseconds
  - Q_1 - time quantum 16 milliseconds
  - Q_2 - FCFS
- Scheduling
  - 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 moved to queue Q_1.
  - At Q_1 job is again served FCFS and receives 16 additional milliseconds. If it still does not complete, it is preempted and moved to queue Q_2.

Multilevel Feedback Queues



Multiple-Processor Scheduling

- CPU scheduling more complex when multiple CPUs are available.
- *Homogeneous processors* within a multiprocessor.
- *Load sharing*
- *Asymmetric multiprocessing* - only one processor accesses the system data structures, alleviating the need for data sharing.

Real-Time Scheduling

Hard real-time systems required to complete a critical task within a guaranteed amount of time.

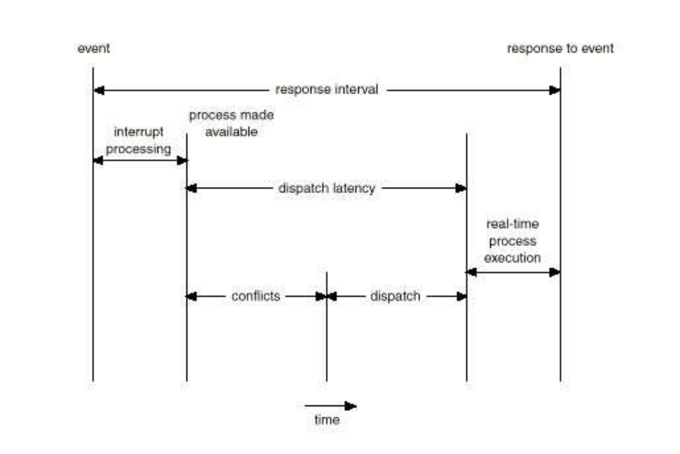
- Resource Reservation
- Hardware support

Soft real-time computing - requires that critical processes receive priority over less fortunate ones.

Dispatch Latency

- Must be small
  - Disable process aging
- System Calls
  - Preemption points on long duration systems call (safe points)
  - Preemptible kernel
- Priority Inversion
  - Priority Inheritance protocol

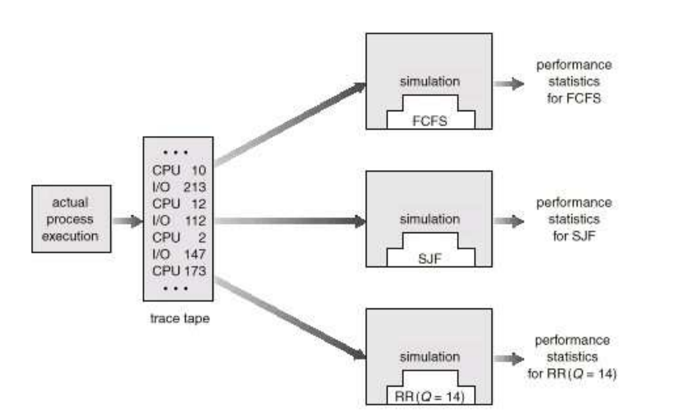
Dispatch Latency



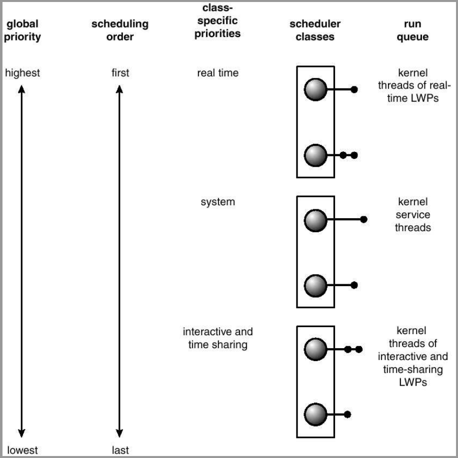
Scheduling Algorithm Evaluation

- Deterministic modeling - takes a particular predetermined workload and defines the performance of each algorithm for that workload.
- Queueing models
  - Aproximate/estimated arrivals and service distributions
- Simulation
- Implementation

Evaluation of CPU Schedulers by Simulation



Solaris 2 Scheduling



Windows 2000 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