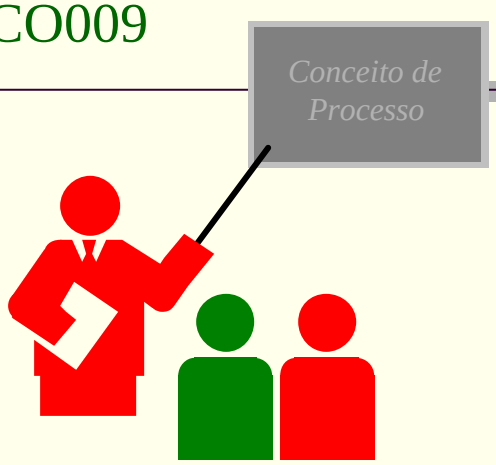
Modulo II
Processos e Threads

Prof. Ismael H F Santos

Ementa

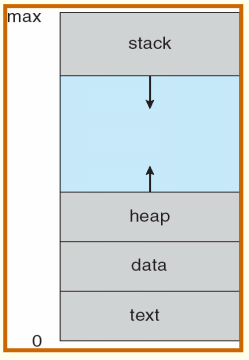
- Introdução aos Sistemas Operacionais
  - *Conceito de Processo*
  - *Subprocesso e Thread*
  - *Escalonamento*
  - *Escalonamento CPU*
    - *FIFO ou FCFS*
    - *SJF*
    - *Coperativo*
    - *Round Robin*
    - *Prioridades*
    - *Múltiplas Filas*
    - *Múltiplas Filas com Realimentação*
    - *Outros Escalonamentos*

SOP – CO009



Conceito de Processo

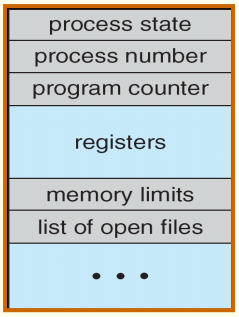
- ***Definição***
  - **Processo** é o ambiente onde se executa um programa. Um mesmo programa pode produzir resultados diferentes, em função do **Processo** no qual ele é executado.
  - Processo pode ser definido também como um programa em execução.
  - A process includes:
    - program counter
    - stack
    - data section

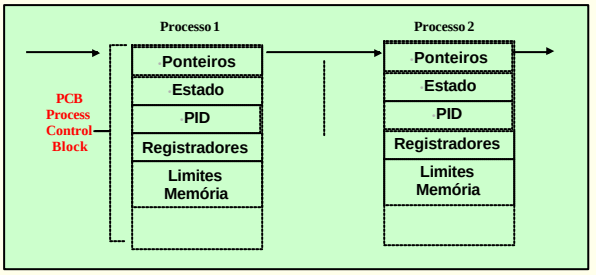


Conceito de Processo

- ***Definição***
  - O SO materializa o processo através de uma estrutura chamada **bloco de controle do processo (Process Control Block PCB).** A partir do **PCB**, o SO mantém todas as informações sobre o processo, como:
    - identificação
    - prioridade
    - estado corrente
    - recursos alocados
    - informações sobre o programa em execução, Program Counter, CPU registers.
    - CPU scheduling information
    - Memory-management information
    - Accounting information
    - I/O status information

Conceito de Processo



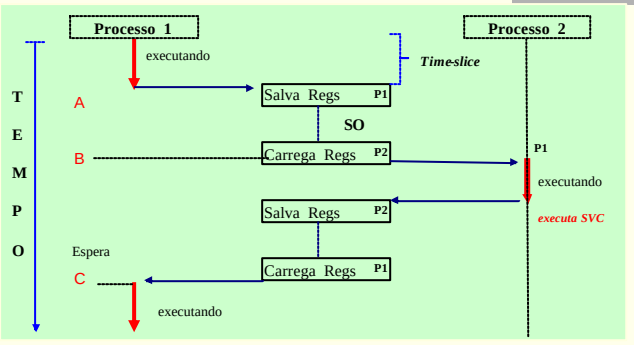


- O processo pode ser dividido em três elementos básicos: contexto de hardware (chw), contexto de software (csw) e espaço de endereçamento (ee), que juntos mantêm todas as informações necessárias à execução do programa.

Conceito de Processo

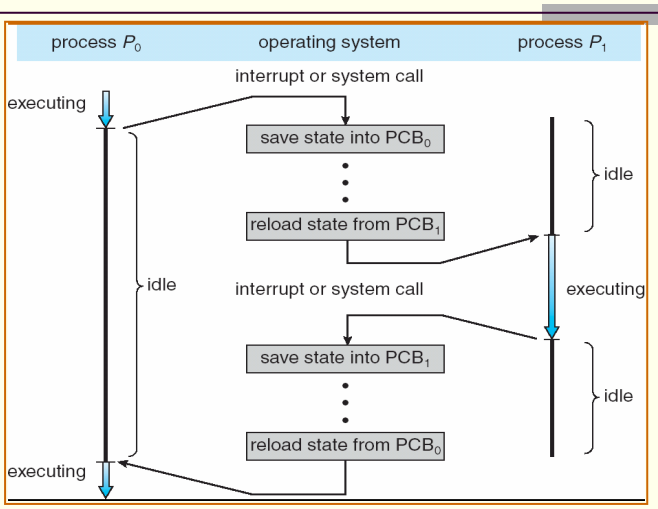
- **Contexto de HW (CHW)**
  - *O **Contexto de Hardware** constitui-se, basicamente, do conteúdo de registradores: program counter (PC), stack pointer (SP) e bits de estado. Quando um processo está em execução, o seu contexto de hardware está armazenado nos registradores do processador. No momento em que o processo perde a utilização da UCP, o sistema salva suas informações no seu **CHW.***
  - *O **CHW** é fundamental para a implementação dos SOs de tempo compartilhado onde os processos se revezam na utilização do processador. podendo ser interrompidos e, posteriormente, restaurados como se nada tivesse acontecido. A troca de um processo por outro na UCP, realizada pelo SO, é notificada através da **mudança de contexto** (**context switching**).*

Salvamento de Contexto



- *A mudança de contexto consiste em salvar o conteúdo dos registradores da UCP e carregá-los com os valores referentes ao do processo que esteja ganhando a utilização do processador.*

Salvamento de Contexto



Conceito de Processo

- ***Contexto de SW (CSW)***
  - *O **Contexto de Software** especifica características do processo que vão influir na execução de um programa. como, por exemplo, o número máximo de arquivos abertos simultaneamente ou o tamanho do buffer para operações de E/S. Essas características são determinadas no momento da criação do processo, podendo algumas ser alteradas durante sua existência.*

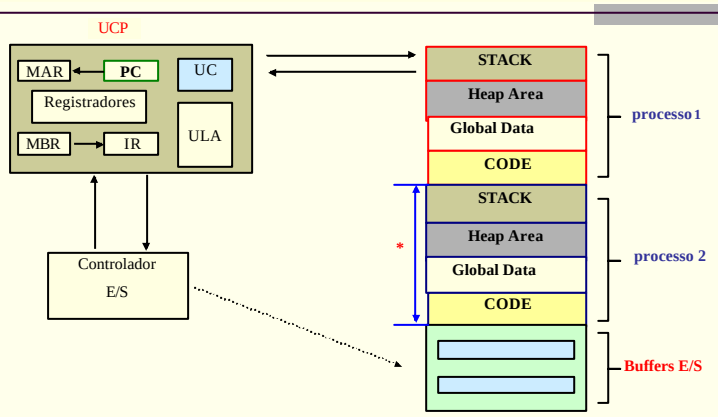
Conceito de Processo

- ***Contexto de SW (CSW)***
  - *O **CSW** define três grupos de informações sobre um processo:*
    1. ***Identificação - PID - process identification; UID - user identif.**, atribuídas ao processo no momento de sua criação.*
    2. ***Quotas -** As quotas são os limites de cada recurso do sistema que um processo pode alocar.*
    3. ***Privilégios -** Os privilégios definem o que o processo pode ou não fazer em relação ao sistema e aos outros processos.*

Conceito de Processo

- ***Espaço de Endereçamento***
  - *O espaço de endereçamento é a área de memória do processo onde o programa será executado, além do espaço para os dados utilizados por ele. Cada processo possui seu próprio espaço de endereçamento, que deve ser protegido do acesso dos demais processos.*
  - *Atualmente o modelo mais geral para o processo executando na memória principal é o mostrado a seguir na figura*

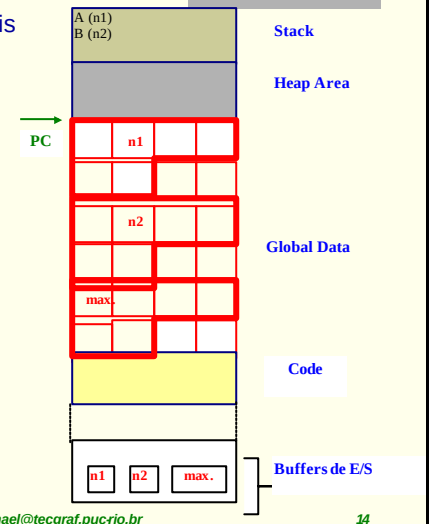
Espaço de Endereçamento



* *Vejamos como é feita a execução de um programa que irá imprimir o maior de dois números dados pelo usuário*

Process Execution

```
Programa MAIOR; {Imprimir o maior de dois
    números}
VAR {global}
    n1, n2, max: integer;
Function maior (A, B: integer): integer;
BEGIN
    IF A > B then
          maior:= A;
    ELSE
          maior:= B;
END;
BEGIN {pgm principal}
    READLN (n1); READLN (n2);     max:=
    maior (n1, n2);
    WRITELN ('maximo entre', n1, n2, 'é',
    max);
END
```



Conceito de Processo

- ***Estados de um Processo***
  - *Um processo durante a sua existência passa por uma série de estados:*
    - ***Execução (running) -*** *quando está sendo processado pela UCP. Em sistemas com apenas um processador, somente um processo pode estar sendo executado num dado instante de tempo. Os processos se revezam na utilização do processador segundo uma política estabelecida pelo sistema operacional. Já em sistemas com múltiplos processadores. vários processos podem estar sendo executado ao mesmo tempo, dependendo do número de processadores. Existe também a possibilidade de um mesmo processo ser executado por mais de um processador (processamento paralelo).*

Conceito de Processo

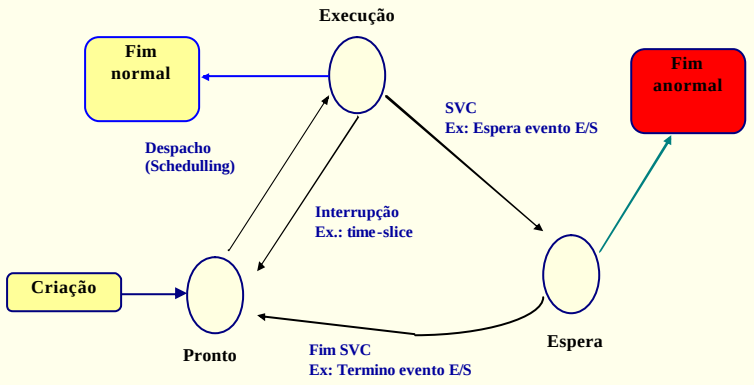
- ***Estados de um Processo***
  - ***Pronto (ready) -*** *Um processo está no* ***estado de Pronto*** *quando apenas aguarda uma oportunidade para executar, ou seja, espera que o sistema operacional aloque a UCP para sua execução. O sistema operacional é responsável por determinar a ordem pela qual os processos em estado de pronto devem ganhar a UCP. Normalmente existem vá-rios processos no sistema no estado de pronto.*

Conceito de Processo

- ***Estados de um Processo***
  - ***Espera (wait) -*** *Um processo está no* ***estado de Espera*** *quando aguarda algum evento externo ou por algum recurso para poder prosseguir seu processamento. Como exemplo. podemos citar o término de uma operação de entrada/saída ou a espera de uma deter-minada data e/ou hora para poder continuar sua execução.*

Conceito de Processo

- ***Diagrama de Transição de Estados***

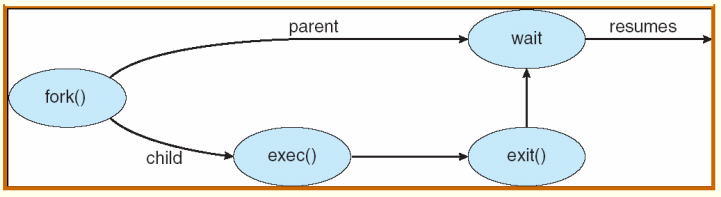


Process Creation

- Parent process create children processes, which, in turn create other processes, forming a tree of processes
- Resource sharing
  - Parent and children share all resources
  - Children share subset of parent's resources
  - Parent and child share no resources
- Execution
  - Parent and children execute concurrently
  - Parent waits until children terminate

Process Creation (Cont.)

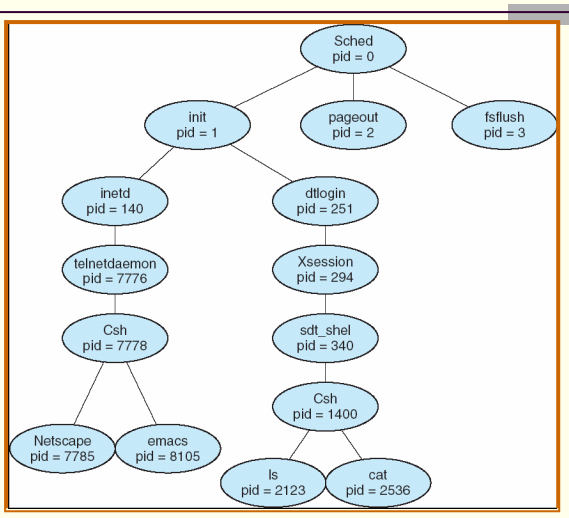
- Address space
  - Child duplicate of parent
  - Child has a program loaded into it
- UNIX examples
  - **fork** system call creates new process
  - **exec** system call used after a **fork** to replace the process' memory space with a new program



C Program Forking Separate Process

```
int main() {
    pid_t  pid = fork();          /* fork another process */
    if (pid < 0) { /* error occurred */
            fprintf(stderr, "Fork Failed");
            exit(-1);
    }
    else if (pid == 0) { /* child process */
            execlp("/bin/ls", "ls", NULL);
    }
    else { /* parent process */
            /* parent will wait for the child to complete */
            wait (NULL);
            printf ("Child Complete");
            exit(0);
    }
}
```

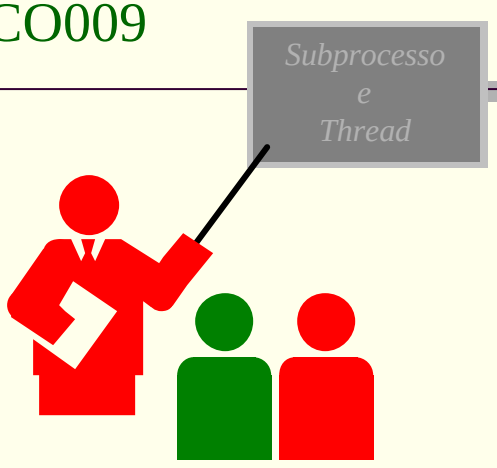

A tree of processes on a typical Solaris



Process Termination

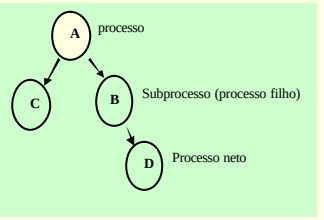
- Process executes last statement and asks the operating system to delete it (**exit**)
  - Output data from child to parent (via **wait**)
  - Process' resources are deallocated by operating system
- Parent may terminate execution of children processes (**abort**)
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - If parent is exiting
    - Some operating system do not allow child to continue if its parent terminates
      - All children terminated - *cascading termination*

SOP – CO009



Conceito de Processo

- ***Subprocesso e Thread***
  - *Um processo pode criar outros processos de maneira hierárquica. Quando um processo (processo pai) cria um outro, chamamos o processo criado de **subprocesso** ou processo filho. O **subprocesso**, por sua vez, pode criar outros **subprocessos**.*



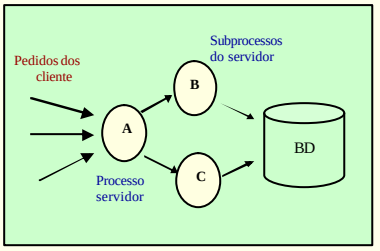
  - *A utilização de **subprocessos** permite dividir uma aplicação em partes que podem trabalhar de forma concorrente.*

Conceito de Processo

- ***Subprocesso e Thread***

Exemplo:

Suponha que um processo seja responsável pelo acesso a um banco de dados e existam vários usuários solicitando consultas sobre esta base.



Caso um usuário solicite um relatório impresso de todos os registros, os demais usuários terão de aguardar até que a operação termine. Com o uso de subprocessos. cada solicitação implicaria a criação de um novo processo para atendê-la, aumentando o throughput da aplicação e, conseqüentemente, melhorando seu desempenho.

Concorrência em arquitetura cliente cliente-servidor: servidor

- Atendimento simultâneo a vários clientes

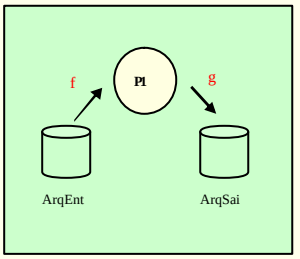
Concorrência em arquitetura cliente cliente-servidor: cliente

- Melhor estrutura da aplicação:
  - resposta a eventos de interface e de rede
- Melhor aproveitamento do tempo:
  - disparo de diversas solicitações simultâneas
  - tratamento local de dados enquanto espera resultado de solicitação

Conceito de Processo

- ***Subprocesso e Thread***
  - *Exemplo: Cópia de Arquivos*

```
Assign(f, ‘ArqEnt´);
Assign(g, ‘ArqSai´);
Reset (f,ArqEnt);
Rewrite (g,ArqSai);
Read (f,Reg);
While not eof(f) do
 Begin
   Write (g, reg);
   Read (f, g);
 End
```

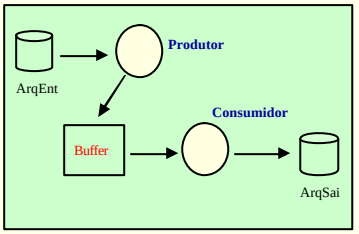


-> *Será que podemos melhorar este programa ?*

Conceito de Processo

- ***Subprocesso e Thread***

Uma solução possível seria a criação de dois processos, um processo chamado Produtor que se encarregará de ler o arquivo de entrada e carregar um Buffer intermediário e um segundo subprocesso chamado Consumidor que irá ler as informações do Buffer e gravará os dados no arquivo de saída.



A função do Buffer é a de prover um meio de armazenamento para os dois processos, de tal forma que se o Buffer for infinito ambos nunca ficarão bloqueados (Explique).

Conceito de Processo

Subprocesso e Thread

- O uso de subprocessos no desenvolvimento de aplicações concorrentes demanda consumo de diversos recursos do sistema. Sempre que um novo processo é criado o SO deve alocar recursos (contexto de HW, contexto de SW e espaço de endereçamento) para cada processo além de consumir tempo de UCP neste trabalho. No caso de término do processo, o sistema desperdiça tempo para desalocar recursos previamente alocados.

- Na tentativa de diminuir o tempo gasto na criação/eliminação de processos, bem como economizar recursos do sistema como um todo, foi introduzido o conceito de thread (ou processo leve ou linha de controle ou linha de execução).

Conceito de Processo

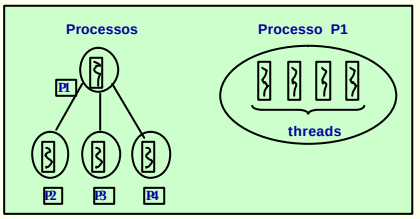
Subprocesso e Thread

- Em um SO com Kernel com capacidade para criar múltiplas threads (multithreaded Kernel) não é necessária a criação de vários processos para se implementar aplicações concorrentes. Em um SO Multithread cada processo pode responder a várias solicitações concorrentes.

- Threads compartilham o processador da mesma maneira que um processos. Por exemplo, enquanto uma thread espera por uma operação de E/S, outra thread pode estar executando. Cada thread possui seu próprio conjunto de registradores (contexto de HW) , porém compartilha o mesmo espaço de endereçamento com as demais threads do processo.

Conceito de Processo

Subprocesso e Thread



- Na figura ao lado existem quatro processos, cada um com seu próprio contexto de HW, contexto de SW e espaço de endereçamento; e um único processo com 3 threads de execução, cada uma com seu próprio contexto de HW e contexto de SW.

- O mecanismo de runtime é responsável pelo despacho para execução da thread de maior prioridade, ou da thread que estava esperando o fim de alguma operação de E/S. O controle de execução de threads é feito de forma cooperativa.

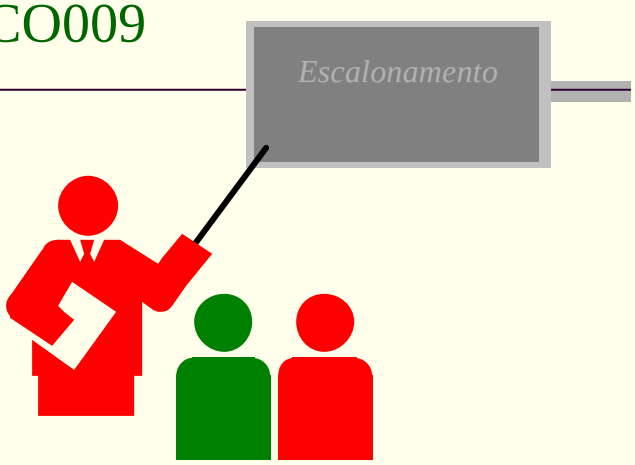
Conceito de Processo

Subprocesso e Thread

- Quando uma thread está sendo executada o contexto de HW da respectiva thread é carregado no processador. No momento em que uma thread perde (fim de time-slice) ou libera (yield) a UCP, o SO salva informações. Threads passam pelos mesmos estados que passam os processos.

- A grande diferença entre subprocesso e thread é em relação ao espaço de endereçamento. Enquanto subprocessos possuem, cada um, espaços independentes e protegidos, threads compartilham o mesmo espaço de endereçamento do processo, sem nenhuma proteção, permitindo assim que uma thread possa alterar dados de outra thread.

SOP – CO009



Gerência do Processador

- ***Escalonamento (Scheduling)***

  O conceito básico que gerou a implementação de sistemas multiprogramáveis foi a necessidade da UCP ser compartilhada entre os diversos processos. Para isso tornou-se necessário a adoção de um critério para determinar qual a ordem na escolha dos processos para execução dentre os vários que concorrem pela utilização do processador. A este critério denominamos ***Escalonamento****.*

Gerência do Processador

- ***Objetivos do Escalonamento***
  - manter a UCP ocupada a maior parte do tempo;
  - maximizar o throughput;
  - oferecer tempos de respostas aceitáveis para usuários interativos.
- ***Criterios de Escalonamento***
  - Utilização da UCP:
  - Throughput;
  - Tempo de turnaround;
  - Tempo de Resposta.

Gerência do Processador

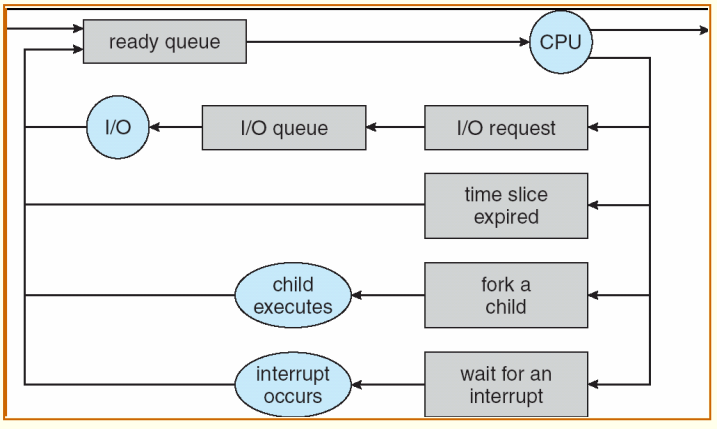
- O algoritmo de escalonamento busca otimizar a utilização da UCP e o Throughput, enquanto tenta diminuir os tempos de turnaround e de resposta.

- O algoritmo de escalonamento não é o único responsável pelo tempo de execução de um processo. Outros fatores, como o tempo de processamento (tempo de UCP) e de espera em operações de E/S. devem ser considerados no tempo total da execução (tempo de parede ou elapsed time ou wall clock time). O escalonamento somente afeta o tempo de espera de processos na fila de pronto.

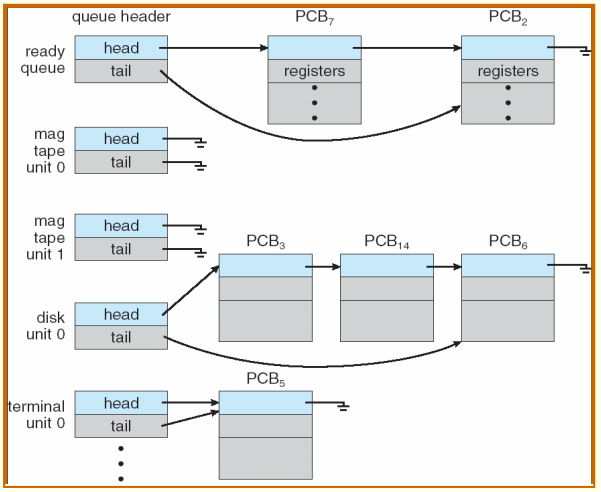
Process Scheduling Queues

- **Job queue** – set of all processes in the system
- **Ready queue** – set of all processes residing in main memory, ready and waiting to execute
- **Device queues** – set of processes waiting for an I/O device
- Processes migrate among the various queues

Representation of Process Scheduling



Ready Queue And Various I/O Device Queues

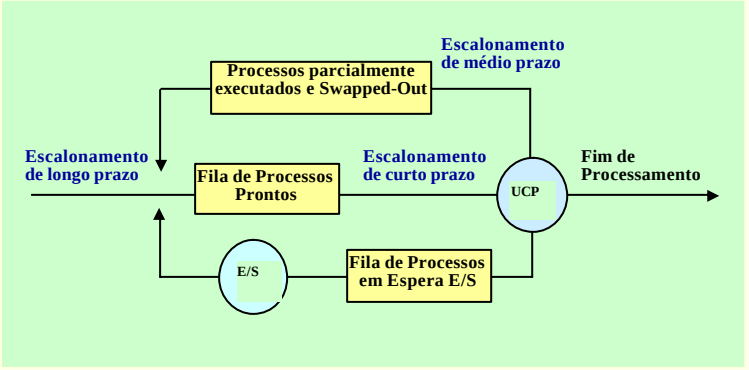


Schedulers

- **Long-term scheduler** (or job scheduler) – selects which processes should be brought into the ready queue
- **Short-term scheduler** (or CPU scheduler) – selects which process should be executed next and allocates CPU

Gerência do Processador

- *Escalonamentos de Longo, Médio e Curto Prazo*



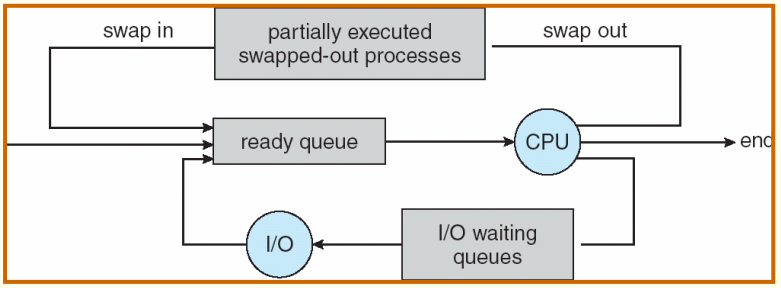
Gerência do Processador

· *O escalonamento de **Longo Prazo** determina quais os jobs serão admitidos pelo SO para processamento.*

· *O de **Curto Prazo** seleciona o próximo processo a executar, dentre todos os processos da fila de processos prontos que estejam residentes em memória.*

· *O de **Médio Prazo**, mais comum em sistemas com **Memória Virtual** , tem objetivo de liberar espaço na memória, removendo processos que estejam esperando algum evento externo (por exemplo, o fim de uma operação de E/S) para area de **Swap**, para permitir que novos processos sejam executados.*

Addition of Medium Term Scheduling



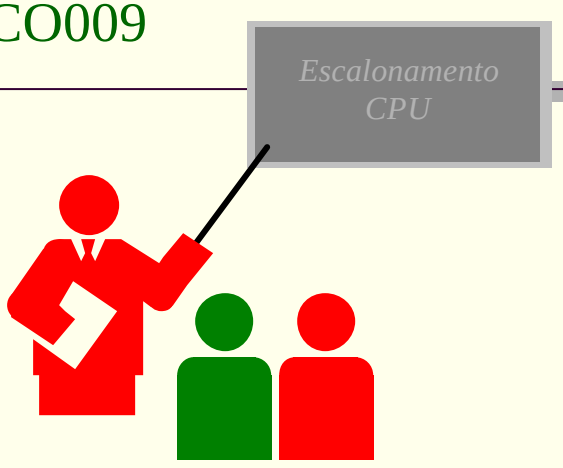
Schedulers (Cont.)

- Short-term scheduler is invoked very frequently (milliseconds) ⇒ (must be fast)
- Long-term scheduler is invoked very infrequently (seconds, minutes) ⇒ (may be slow)
- The long-term scheduler controls the *degree of multiprogramming*
- Processes can be described as either:
  - **I/O-bound process** – spends more time doing I/O than computations, many short CPU bursts
  - **CPU-bound process** – spends more time doing computations; few very long CPU bursts

Context Switch

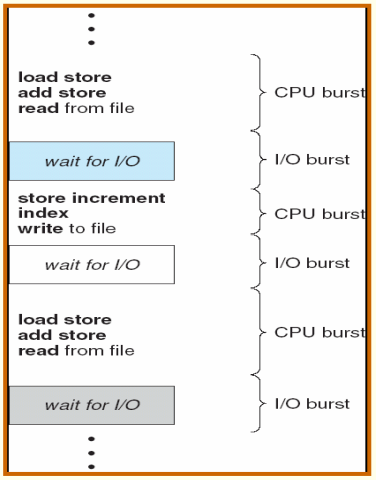
- When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process
- Context-switch time is overhead; the system does no useful work while switching
- Time dependent on hardware support

SOP – CO009

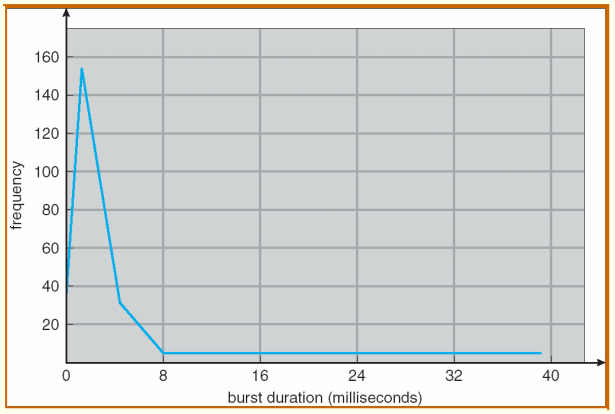


Basic Concepts

- 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



Histogram of CPU-burst Times



CPU Scheduler

- 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
  2. Switches from running to ready state
  3. Switches from waiting to ready
  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
- 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

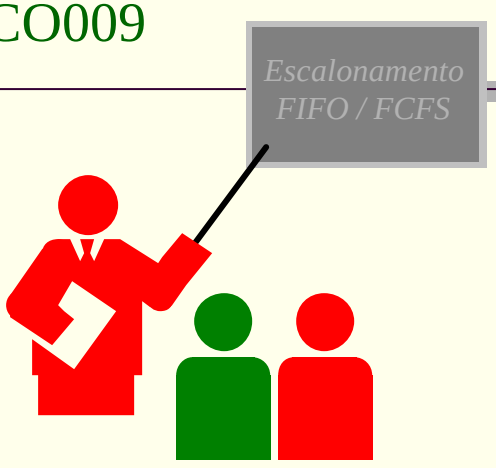
Escalonamento CPU

- ***Tipos de Escalonamentos***

  Não Preemptivos: Implementados inicialmente nos primeiros SOs, onde predominava tipicamente o processamento batch. Neste tipo de escalonamento, quando um processo ganha o direito de utilizar a UCP nenhum outro processo pode lhe tirar este recurso.

  Preemptivos: quando o sistema pode interromper um processo em execução, para que outro utilize o processador. Em sis-temas que não implementam **preempção**. um processo pode utilizar o processador enquanto for necessário.

SOP – CO009



Escalonamento CPU

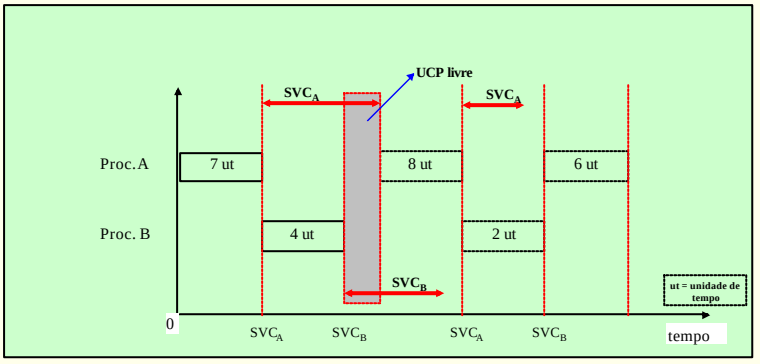
- ***Escalonamento Circular Simples - FIFO ou FCFS***

  · *Todos os processos começam a executar segundo a ordem que são chamados para execução. Quando um processo ganha o processador, ele utilizará o processador até o seu final sem ser interrompido. No caso de ser executada uma SVC, o processo, após ter sido atendida a SVC, voltará para o final da fila de processos prontos.*

  · *O problema do **escalonamento FIFO** é a impossibilidade de se prever quando um processo terá sua execução iniciada, já que isso varia em função do tempo de execução dos processos que se encontram na sua frente.*

Escalonamento CPU

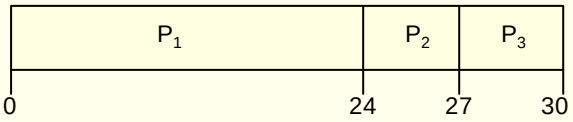
- ***Escalonamento FIFO (FCFS)***



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:



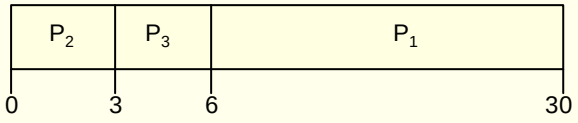
- Waiting time for P_1 = 0; P_2 = 24; P_3 = 27
- Average waiting time: (0 + 24 + 27)/3 = 17

FCFS Scheduling (Cont.)

Suppose that the processes arrive in the order

P_2 , P_3 , P_1

- The Gantt chart for the schedule is:

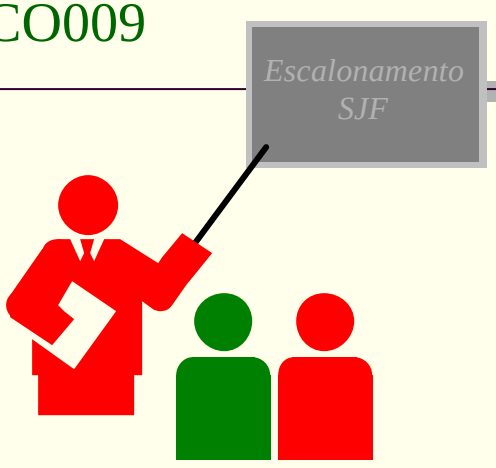


- Waiting time for P_1 = 6; P_2 = 0; P_3 = 3
- Average waiting time: (6 + 0 + 3)/3 = 3
- Much better than previous case !!!
- *Convoy effect* short process behind long process

Escalonamento CPU

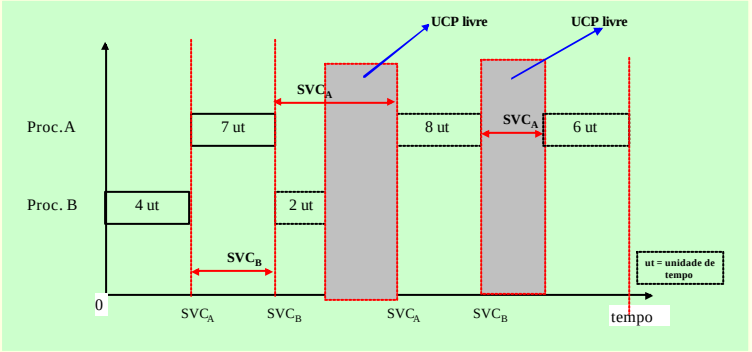
- **Escalonamento Shortest Job First – SJF**
  - *Neste escalonamento cada processo tem associado o seu tempo de execução. Desta forma quando a UCP está livre o processo em estado de pronto que tiver menor tempo de execução será selecionado para execução.*
  - *O escalonamento **SJF** favorece os processos que executam programas menores, além de reduzir o tempo médio de espera (na fila de processos prontos) em relação ao escalonamento FIFO. A dificuldade é determinar, exatamente, quanto tempo de UCP cada processo necessita para terminar seu processamento.*

SOP – CO009



Escalonamento CPU

- **Escalonamento Shortest Job First - SJF**



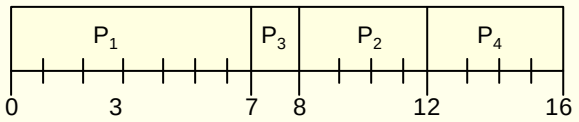
Shortest-Job-First (SJF) Scheduling

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

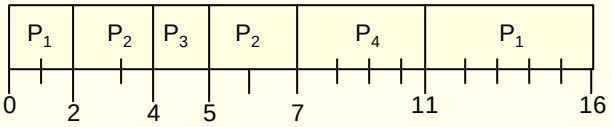


- Average waiting time = (0 + 6 + 3 + 7)/4 = 4

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)



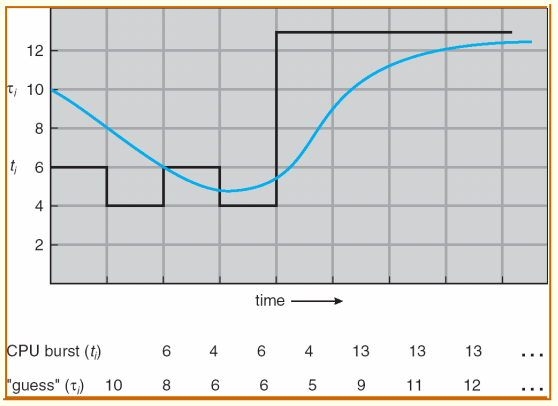
- Average waiting time = (9 + 1 + 0 +2)/4 = 3

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

1. t_n = actual lenght of n^{th} CPU burst
2. t_{n+1} = predicted value for the next CPU burst
3. $a, 0 \le a \le 1$
4. Define : $t_{n=1} = a\ t_n + (1-a)t_n$.

Prediction of the Length of the Next CPU Burst



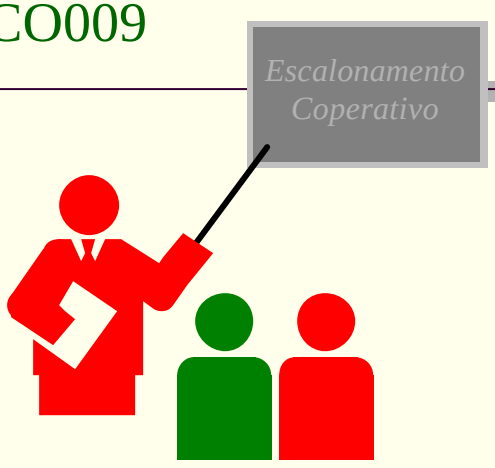
Examples of Exponential Averaging

- $\alpha = 0$
  - $\tau_{n+1} = \tau_n$
  - 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

SOP – CO009



Escalonamento CPU

- ***Escalonamento Cooperativo***
  - *No escalonamento **Cooperativo** alguma política não-preemptiva deve ser adotada. A partir do momento que um processo está em execução, este voluntariamente libera o processador, retornando para a fila de pronto. Sua principal característica está no fato de a liberação do processador ser uma tarefa realizada exclusivamente pelo processo em execução, que de uma maneira cooperativa libera a UCP para um outro processo.*
  - *Neste escalonamento não existe nenhuma intervenção do SO na execução do processo. Isto pode ocasionar sérios problemas na medida em que um programa pode não liberar o processador ou um programa mal escrito pode entrar em looping, monopolizando desta forma a UCP.*

SOP – CO009

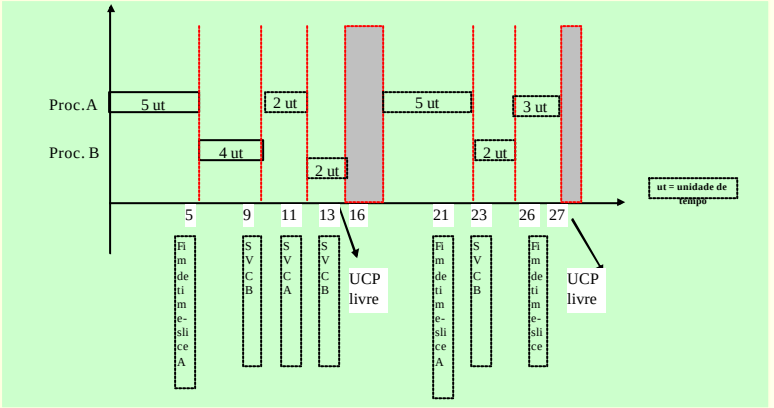
Escalonamento Round Robin

Escalonamento CPU

- ***Escalonamento Circular - Round-Robin***
  - *Implementado através de um algoritmo projetado especialmente para sistemas de tempo compartilhado.O algoritmo é semelhante ao FIFO, porém, quando um processo passa para o estado de execução, existe um tempo limite para a sua utilização de forma continua. Quando este tempo, denominado **time-slice** ou **quantum**, expira sem que antes a UCP seja liberada pelo processo, este volta ao estado de pronto (**preempção**), dando a vez a outro processo. A fila de processos prontos é tratada como uma fila circular.*

Escalonamento CPU

- ***Escalonamento Circular - Round-Robin***

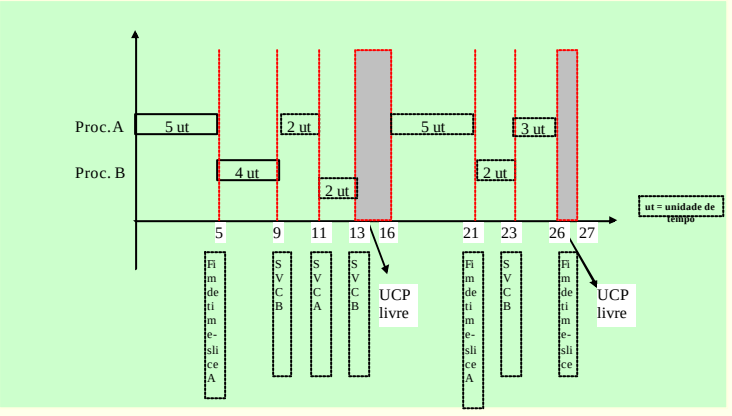


Escalonamento CPU

- ***Escalonamento Circular - Round-Robin***
  - *Algoritmo projetado especialmente para **SOs de Tempo Compartilhado**.O algoritmo é semelhante ao FIFO, porém, quando um processo passa para o estado de execução, existe um tempo limite para a sua utilização de forma continua. Quando este tempo, denominado **time-slice** ou **quantum**, expira sem que antes a UCP seja liberada pelo processo, este volta ao estado de pronto (**preempção por tempo**), dando a vez a outro processo. A fila de processos prontos é tratada como uma fila circular.*

Escalonamento CPU

- ***Escalonamento Circular - Round-Robin***



Escalonamento CPU

- ***Escalonamento Round-Robin***
  - *O **Escalonamento Round-Robin (RR)** consegue melhorar a distribuição de tempo de UCP em relação aos escalonamentos não preemptivos, porém não consegue implementar um compartilhamento eqüitativo entre os diferentes tipos de processos. Isso acontece em razão do escalonamento circular tratar os processos igualmente.*
  - *No **Escalonamento RR** os processos **IO-Bound** são prejudicados em relação aos processos **UCP-Bound**.*

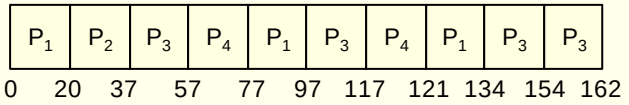
Round Robin (RR)

- 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 ⇒ FIFO
  - *q* small ⇒ *q* must be large with respect to context switch, otherwise overhead is too high !!!

Example of RR, Time Quantum = 20

Process	Burst Time
P_1	53
P_2	17
P_3	68
P_4	24

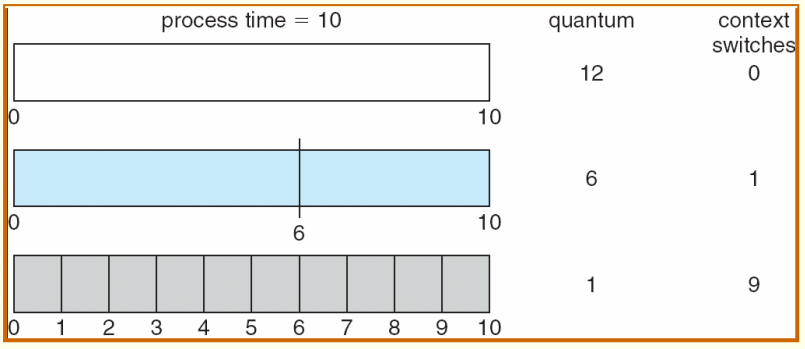
- The Gantt chart is:



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

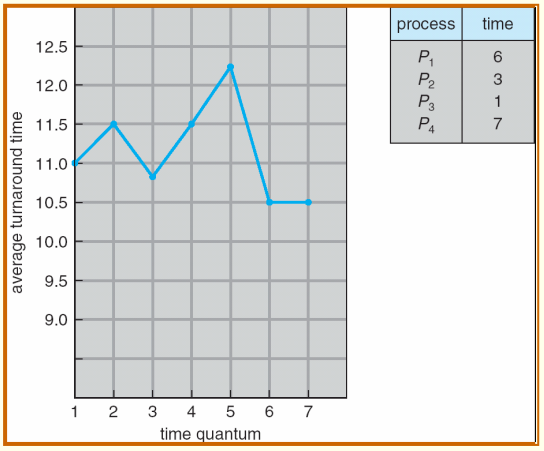
Time Quantum x Context Switch Time

- How a Smaller Time Quantum Increases Context Switches

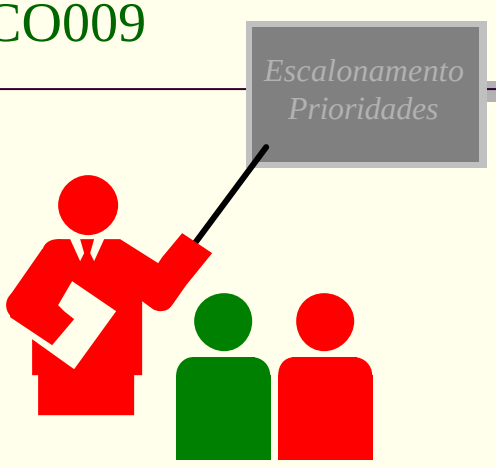


Turnaround Time x Time Quantum

- Turnaround Time Varies With The Time Quantum



SOP – CO009



Escalonamento CPU

Escalonamento por Prioridades

- *Para compensar o excessivo tempo gasto no **estado de espera**, devemos atribuir alguma compensação aos processos **IO-Bound.** Isto pode ser feito através da variação da prioridade de execução associada a cada processo.*

- *No **Escalonamento por Prioridades**, processos de maior prioridade são escalonados preferencialmente. Toda vez que um processo for para a fila de prontos com prioridade superior a do processo em execução, o SO deverá interromper o processo corrente, coloca-lo no estado de pronto e escalonar o processo de maior prioridade para execução. Esse mecanismo é definido como **preempção por prioridade.***

Escalonamento CPU

Escalonamento por Prioridades

- *Assim como na **preempção por tempo** a **preempção por prioridade** é implementada mediante um clock, que interrompe o processador em determinados intervalos de tempo, para que a rotina de **Escalonamento de Curto Prazo** (**Escalonador ou Dispatcher**) reavalie as prioridades e, possivelmente, escalone outro processo.*

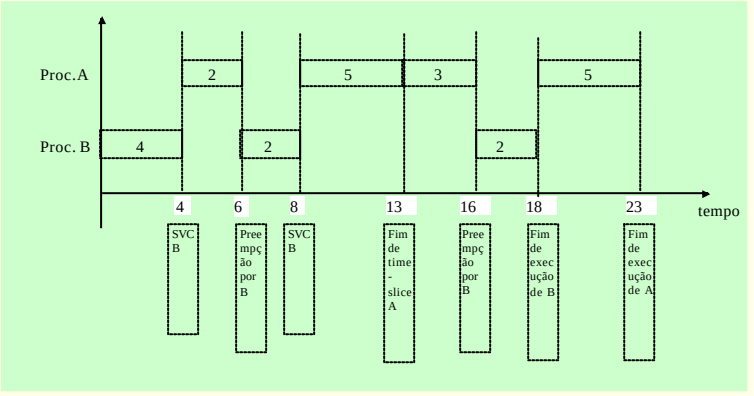
- *A prioridade é uma característica do **contexto de SW** do processo, podendo ser **estática** ou **dinâmica.** A prioridade é dita **estática** quando não é modificada durante a existência do processo. Na prioridade **dinâmica** a prioridade do processo pode ser ajustada de acordo com o tipo de processamento realizado pelo processo e/ou pela carga do sistema.*

Escalonamento CPU

- ***Escalonamento por Prioridades***
  - *Todo processo, ao sair do **estado de espera**, recebe um acréscimop à sua prioridade. Dessa forma, os processos **I/O Bound** terão mais chance de ser escalonados e, assim, compensar o tempo que passam no **estado de espera**. Obeserve que este procedimento não prejudica os processos **CPU Bound**, pois estes podem ser executados enquanto os processos **I/O Bound** esperam por algum evento.*
  - *Um problema potencial é que um processo pode sofrer um **adiamento indefinido** ou **starvation** quando sempre que ele estiver na fila de processos prontos aparecer outro processo de maior prioridade. A utilização de prioridade **dinâmica** tende a diminuir este problema.*

Escalonamento CPU

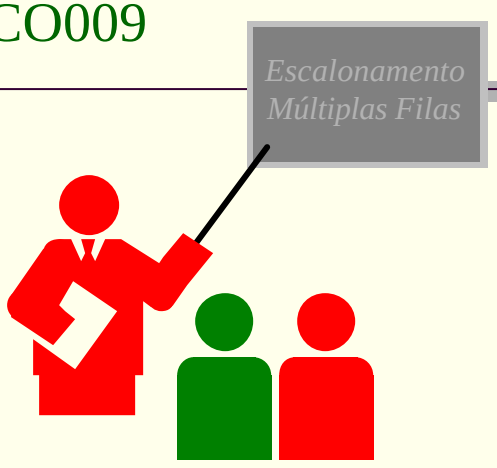
- ***Escalonamento por Prioridades***



Priority Scheduling

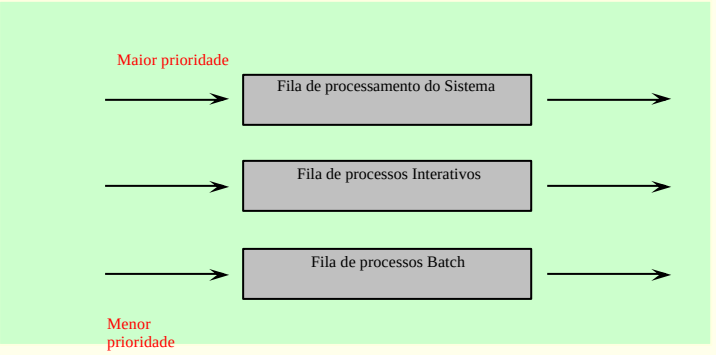
- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer ≡ highest priority)
  - Preemptive
  - nonpreemptive
- SJF is a priority scheduling where priority is the predicted next CPU burst time
- Problem ≡ Starvation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

SOP – CO009



Escalonamento CPU

- ***Escalonamento Múltiplas Filas (Multi-level Queues)***



Escalonamento CPU

- ***Escalonamento Múltiplas Filas – Multi-level Queues***
  - ***Escalonamento por Múltiplas Filas*** *implementa diversas filas de processo no estado de pronto, onde cada processo é associado exclusivamente a uma delas conforme figura anterior.*
  - *Cada fila possui um mecanismo próprio de escalonamento, em função das características do processo. Cada fila possui uma prioridade associada, que estabelece quais filas são prioritárias em relação às outras. O sistema só irá escalonar processos de uma fila se todas as outras filas de prioridade maior estiverem vazias.*

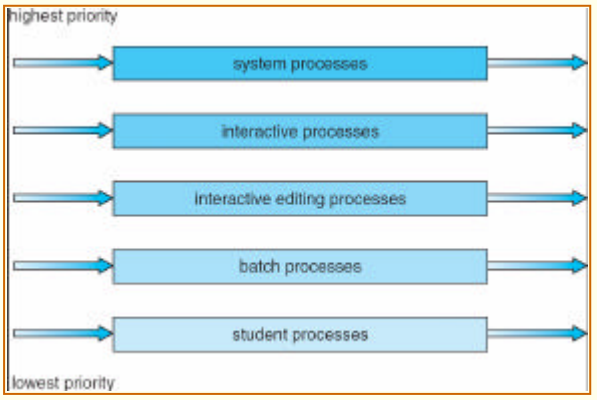
Multilevel Queue

- Ready queue is partitioned into separate queues:
  foreground (interactive)
  background (batch)
- Each queue has its own scheduling algorithm
  - foreground – RR
  - background – FCFS

Multilevel Queue (cont.)

- 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

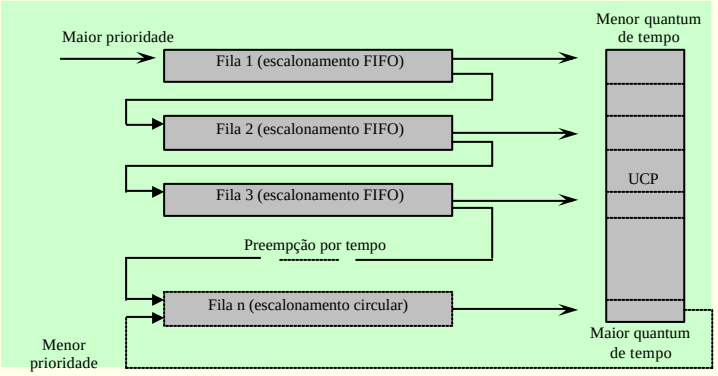


SOP – CO009

Múltiplas Filas com Realimentação

Escalonamento CPU

- ***Escalonamento Múltiplas Filas com Realimentação – FeedBack Multi-level Queues***



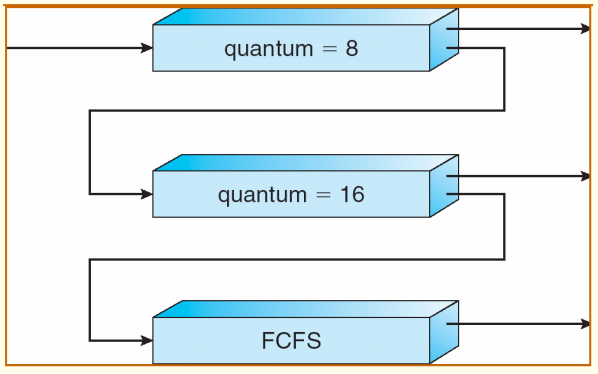
Multilevel Feedback Queue

- 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 – RR with time quantum 8 milliseconds
  - Q_1 – RR time quantum 16 milliseconds
  - Q_2 – FCFS
- Scheduling
  - A new job enters queue Q_0 which is served RR. 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 RR and receives 16 additional milliseconds. If it still does not complete, it is preempted and moved to queue Q_2.

Multilevel Feedback Queues



SOP – CO009

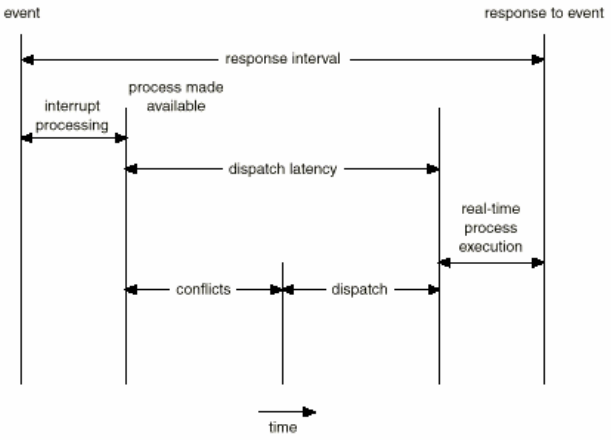
Multiple-Processor Scheduling

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

Dispatch Latency



Thread Scheduling

- Local Scheduling – How the threads library decides which thread to put onto an available LWP
- Global Scheduling – How the kernel decides which kernel thread to run next

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);
```

Pthread Scheduling API

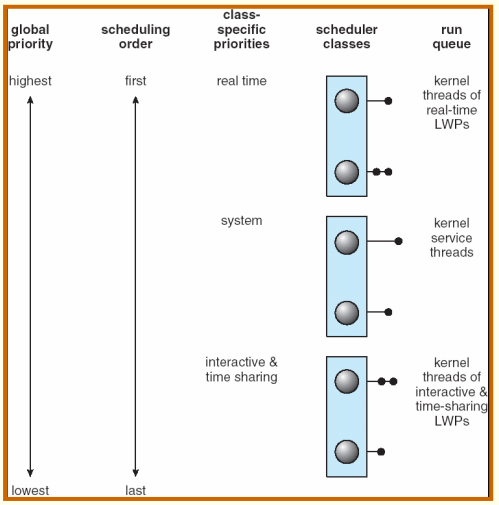
```
    /* create the threads */
    for (i = 0; i < NUM THREADS; i++)
        pthread create(&tid[i], &attr, runner, NULL);

    /* 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) {
    printf("I am a thread\n");
    pthread exit(0);
}
```


Operating System Examples

- Solaris scheduling
- Windows XP scheduling
- Linux scheduling

Solaris 2 Scheduling



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

Windows XP 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

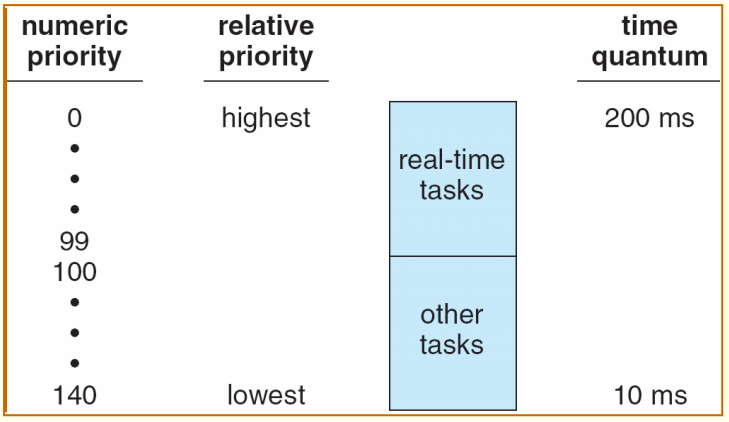
Linux Scheduling

- Two algorithms: time-sharing and real-time
- Time-sharing
  - Prioritized credit-based – process with most credits is scheduled next
  - Credit subtracted when timer interrupt occurs
  - When credit = 0, another process chosen
  - When all processes have credit = 0, recrediting occurs
    - Based on factors including priority and history

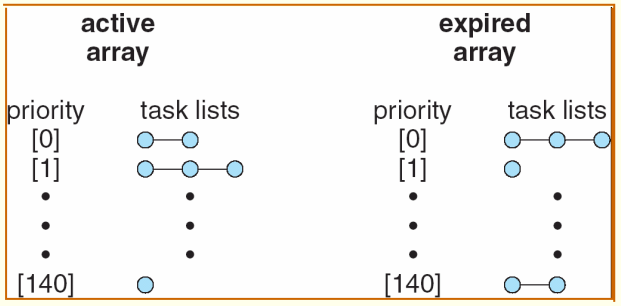
Linux Scheduling (cont.)

- Real-time
  - Soft real-time
  - Posix.1b compliant – two classes
    - FCFS and RR
    - Highest priority process always runs first

The Relationship Between Priorities and Time-slice length



List of Tasks Indexed According to Prorities



Algorithm Evaluation

- Deterministic modeling – takes a particular predetermined workload and defines the performance of each algorithm for that workload
- Queueing models
- Implementation

Evaluation of CPU Schedulers by Simulation