Sistemas Operativos

Licenciatura Engenharia Informática
Licenciatura Engenharia Computacional

Ano letivo 2024/2025

Nuno Lau (nunolau@ua.pt)

Escalonador do CPU

- Selecciona de entre os processos *Ready* qual o que irá ser executado no(s) CPU(s)
- Escalonador é activado quando o processo:
  - Muda do estado de *running* para *waiting*
  - Muda do estado *running* para *ready*
  - Muda do estado *waiting* para *ready*
  - Termina
- Os escalonadores que usam apenas 1 e 4 são designados *non preemptive*
- Escalonadores que usam 2 e 3 são *preemptive*

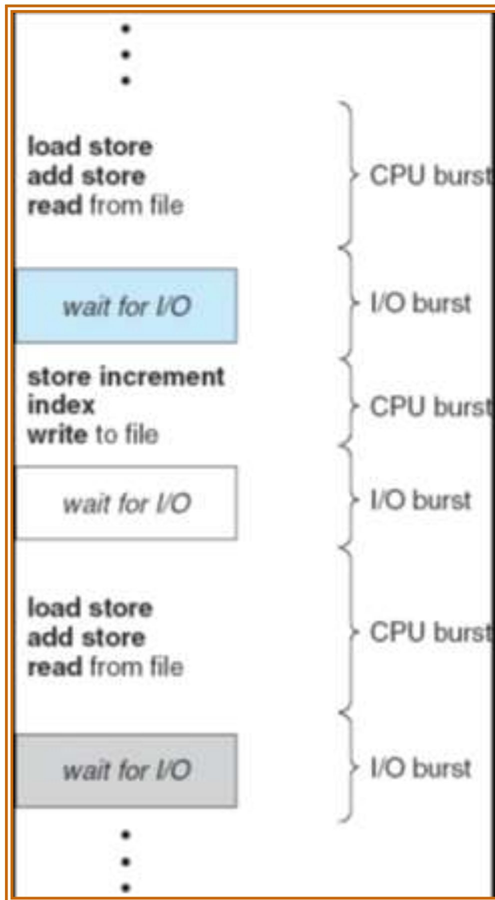
Escalonador do CPU

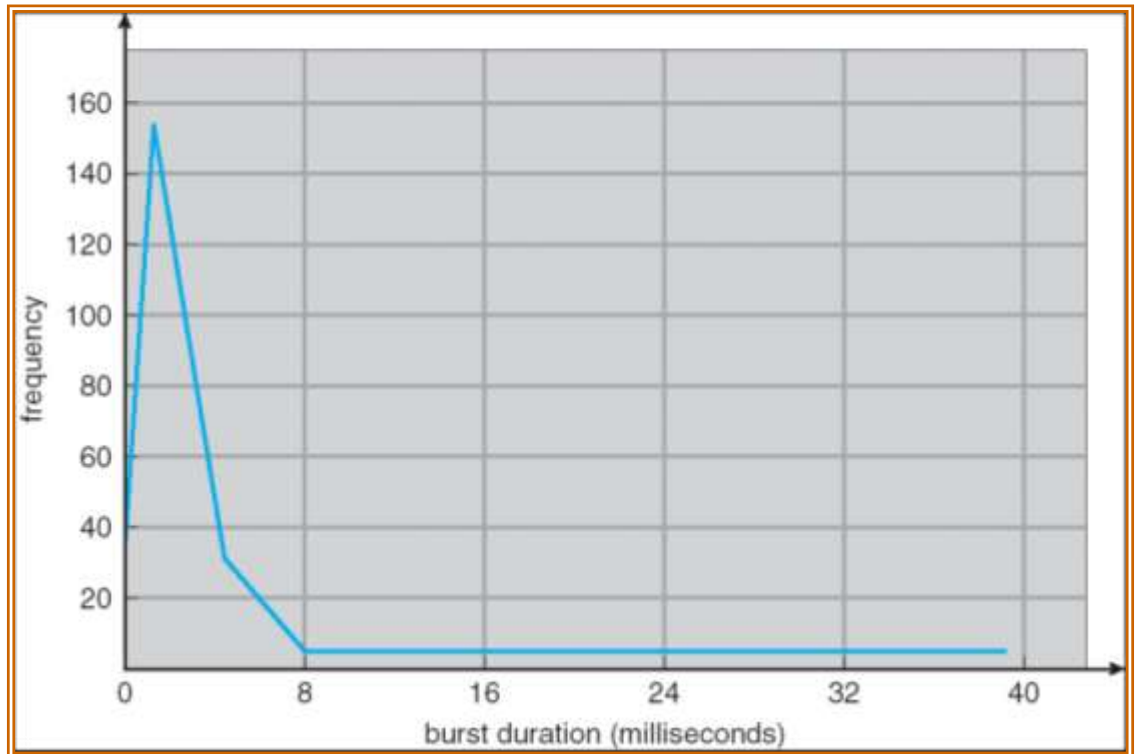
- *Dispatcher* encarrega-se de colocar o processo selecionado pelo escalonador em execução no CPU
  - Mudança de contexto
  - Alterar CPU para modo de utilizador
  - Saltar para instrução do programa que permite continuar a execução do processo selecionado
- *Dispatch latency* – tempo que o *Dispatcher* demora entre parar um processo e reiniciar o processo selecionado pelo escalonador

Avaliação do Escalonamento

- Utilização do CPU
  - Manter CPU ocupado
- Débito
  - número de processos que terminam por unidade de tempo
- Tempo do processo (*turnaround time*)
  - Tempo entre submissão do processo até este terminar
- Tempo de espera
  - Tempo que o processo está à espera no estado Ready
- Tempo de resposta
  - Tempo entre pedido e primeira resposta (eventualmente parcial) a esse pedido

Execução de um processo



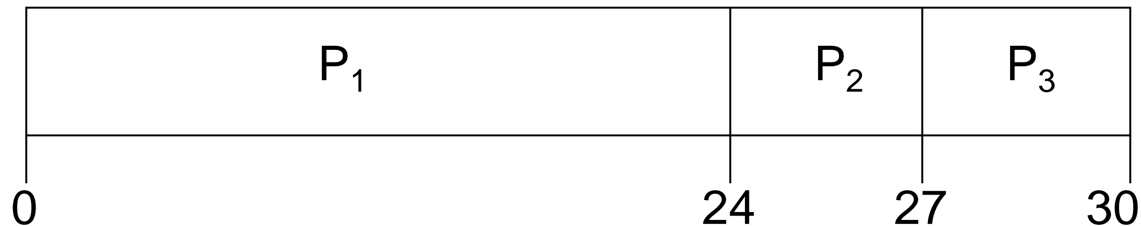


Escalonamento FCFS

- *First-Come, First-Served*

Process	*Burst Time*
P_1	*24*
P_2	*3*
P_3	*3*

- Se os processos chegarem pela ordem 1, 2, 3, então:



- Tempo de espera: $P_1 = 0$; $P_2 = 24$; $P_3 = 27$
- Tempo médio de espera: (0 + 24 + 27)/3 = **17**

Escalonamento FCFS

- Mas se os processos chegarem pela ordem 2, 3, 1, então:

- Tempo de espera: $P_1 = 6$; $P_2 = 0$; $P_3 = 3$
- Tempo médio de espera: (6 + 0 + 3)/3 = **3 !!!**

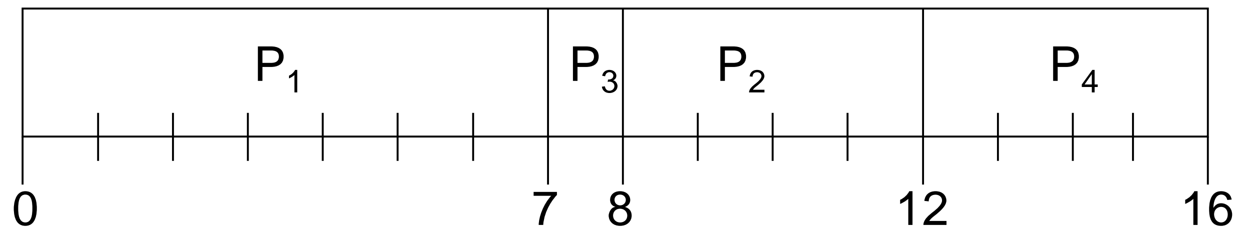
Escalonamento SJF

- *Shortest Job First*
- Ordena os processos considerando a duração do próximo CPU burst. Executa primeiro os processos com CPU burst mais curtos
- Duas opções
  - *Nonpreemptive* – uma vez atribuído o CPU o processo fica em *Running* até terminar o CPU *burst*
  - *Preemptive* – se um processo entra na fila de *Ready* com um CPU Burst menor do que o tempo restante do CPU *burst* do processo em execução, atribuir o CPU ao processo que entrou em *Ready*. Também conhecido como *Shortest-Remaining-Time-First* (SRTF)
- SJF é óptimo do ponto de vista do tempo médio de espera de um conjunto de processos

Escalonamento 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)*

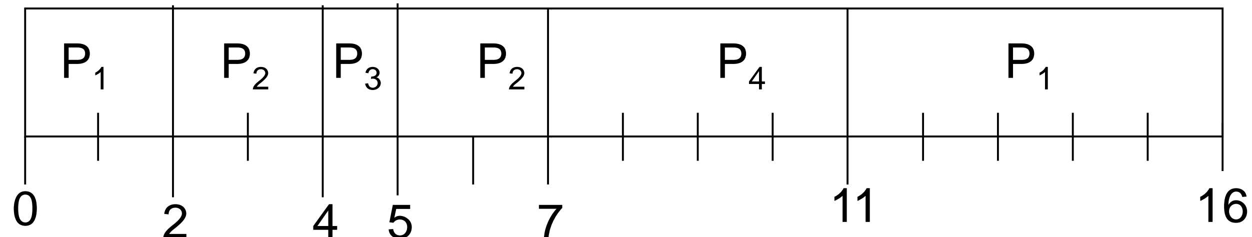


- Tempo médio de espera= (0 + 6 + 3 + 7)/4 = **4**

Escalonamento 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)*



- Tempo médio de espera= (9 + 1 + 0 + 2)/4 = **3**

Determinar o tempo do próximo CPU *Burst*

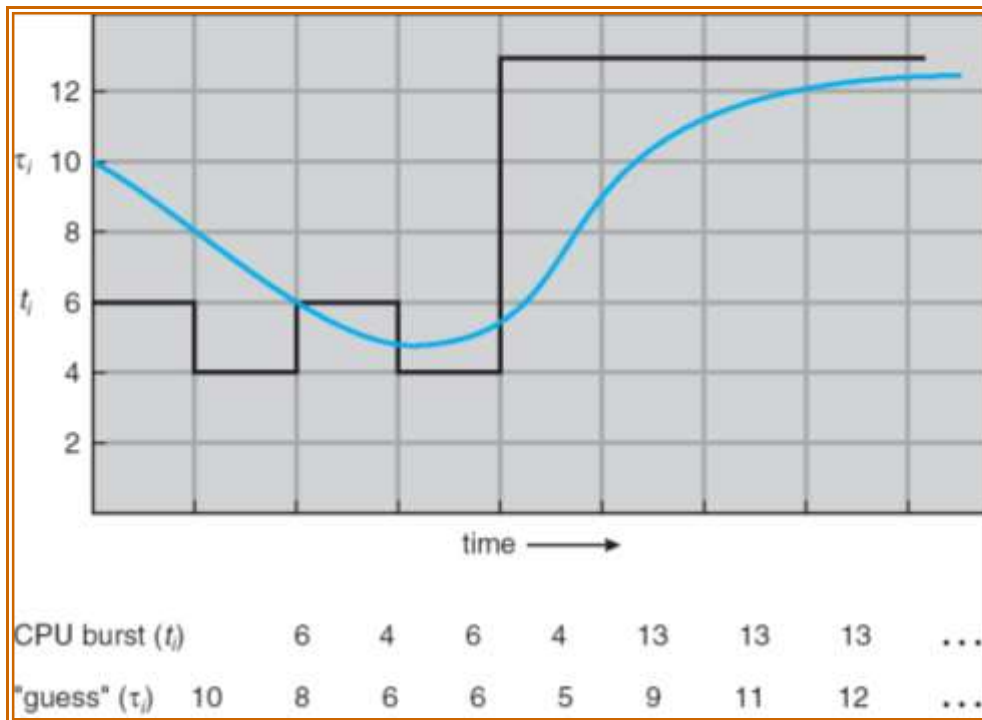
- Os tempos dos CPU *Burst* não são, em geral, conhecidos
- Solução: tentar obter boas estimativas

- Como?
  - Usar histórico do processo para prever o futuro
  - Exemplo: Média exponencial

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

  - t_n é o tempo do último CPU *Burst*, τ é a estimativa do CPU burst

Determinar o tempo do próximo CPU Burst



Escalonamento por prioridades

- *Priority sheduling*
- É associado um nível de prioridade (inteiro) com cada processo
  - Não existe acordo sobre se a prioridade mais alta corresponde a valores baixos ou altos do nível de prioridade
  - Iremos assumir que números baixos representam maior prioridade
- O CPU é atribuido ao processo com maior prioridade
  - Preemptive
  - Nonpreemptive
- SJF é um caso particular de escalonamento por prioridades
- Problema: Adiamento indefinido
  - Processos com prioridade baixa podem nunca executar
- Solução: Contar com o tempo de espera (*aging*)
  - Aumentar a prioridade dos processos em espera à medida que o tempo passa

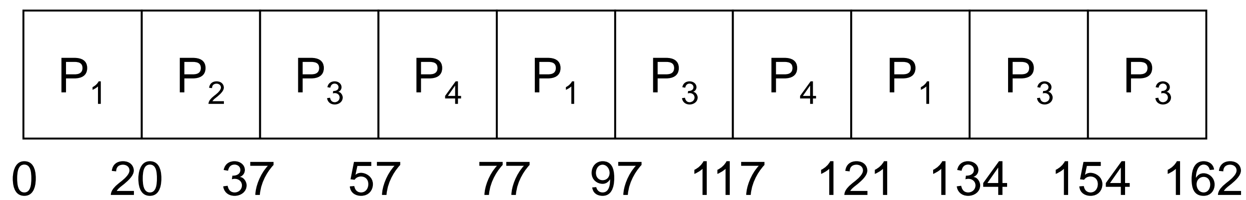
Round Robin

- Versão *Time sharing* e *preemptive* de FCFS
- Cada processo pode usar o CPU, no máximo, por determinado tempo (*time quantum*). Se o processo não bloquear antes do tempo definido é retirado de execução e passa para o fim da lista de Ready
  - *Time quantum* varia, em geral, entre 10 e 100ms
- Se existem n processos na fila de *Ready* (nenhum em execução) e o *time quantum* é q então:
  - cada processo usa cerca de 1/n do processador
  - Um processo nunca espera mais do que (n-1).q unidades de tempo
- Desempenho
  - Q grande $\Rightarrow$ FCFS
  - Q pequeno $\Rightarrow$ o overhead da mudança de contexto pode ser significativo

Round Robin com q=20

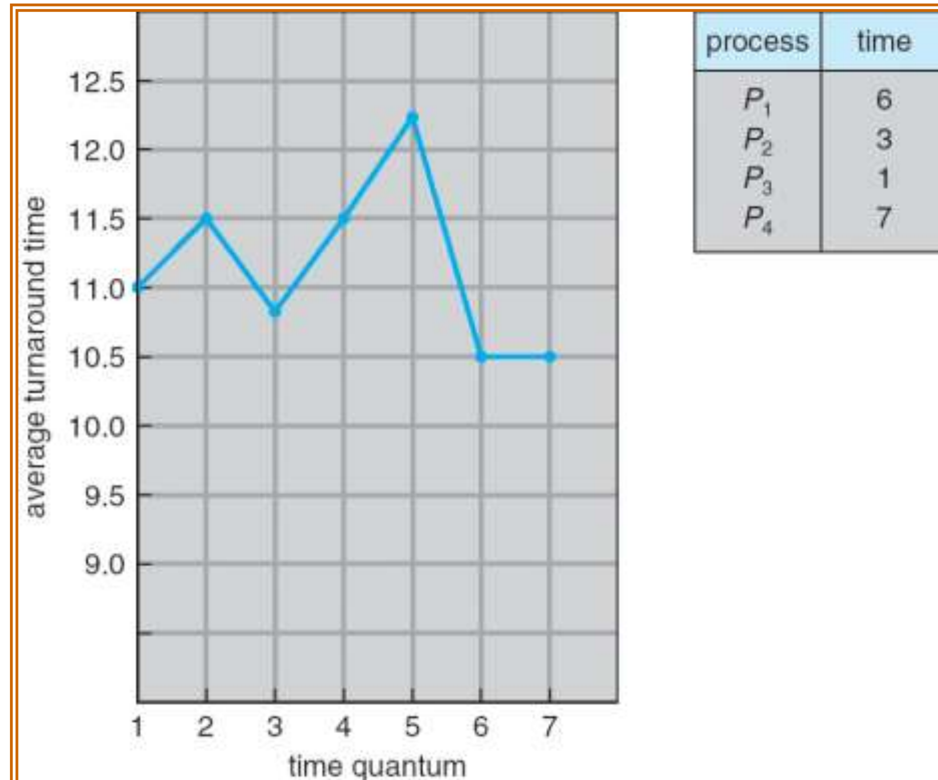
Process	Burst Time
P1	53
P2	17
P3	68
P4	24

- O escalonamento será:



- Tipicamente RR tem maior tempo médio de espera do que SJF, mas melhor tempo de resposta

Tempo do processo médio vs. *Time quantum*

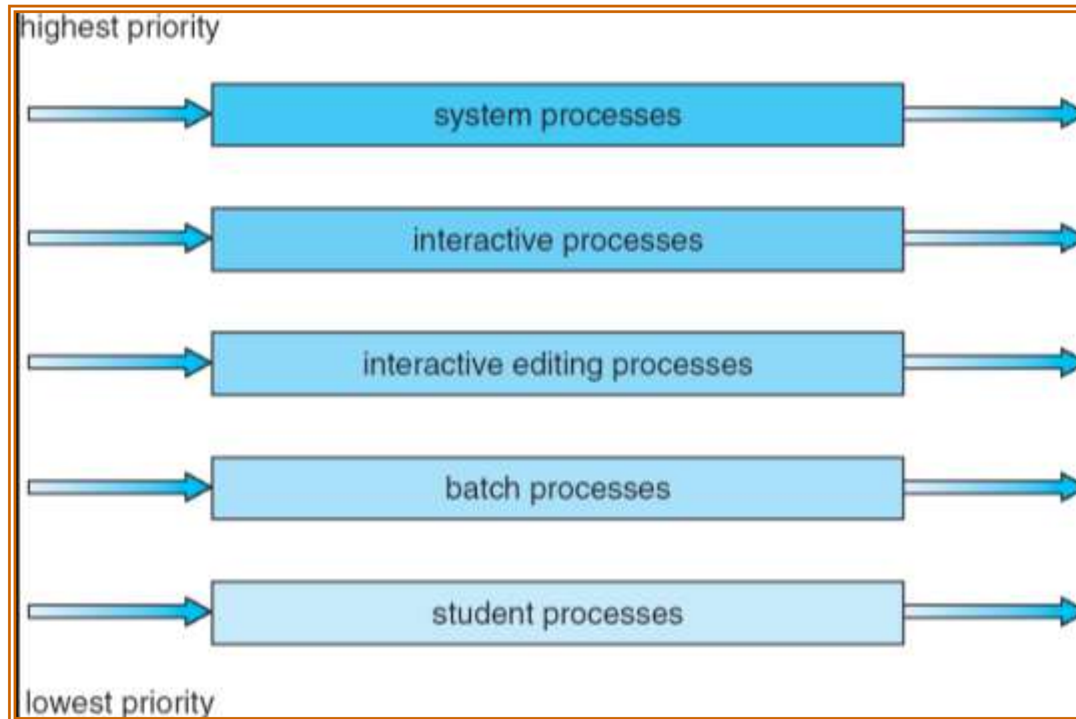


process	time
P_1	6
P_2	3
P_3	1
P_4	7

FIFO multi-nível

- *Multilevel queue*
- Fila de *Ready* é dividida em 2:
  - *Foreground* (interactiva)
  - *Background* (batch)
- Cada Fila tem pode ter a sua politica de escalonamento
  - *Foreground* – RR
  - *Background* – FCFS
- Escalonamento entre as 2 filas
  - Baseado em prioridades fixas
    - Só executar processos em background se fila *Ready* de *foreground* estiver vazia
  - Divisão do tempo (*Time slice*)
    - Cada fila tem um certo tempo de CPU disponível (Ex: 80% RR 20% FCFS)

FIFO multi-nível



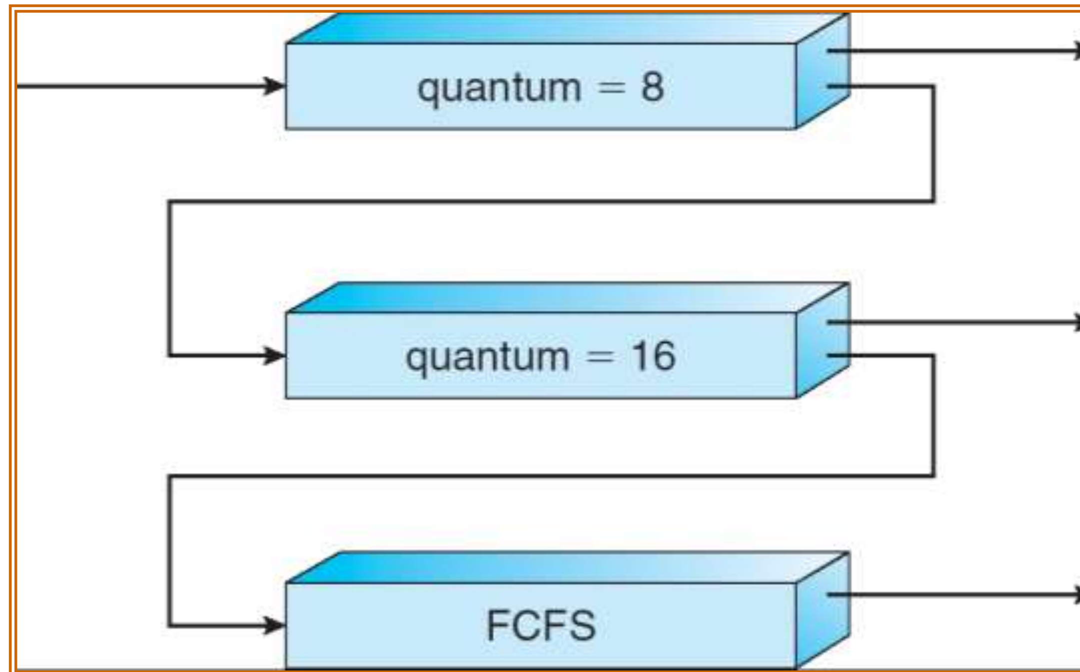
FIFO multi-nível com realimentação

- *Multilevel Feedback Queue*
- Um processo pode mover-se entre as várias filas
- Parâmetros
  - Número de filas
  - Algoritmo de escalonamento de cada fila
  - Algoritmos de elevar e descer a prioridade de um processo
  - Algoritmo de atribuição da prioridade inicial de um processo

FIFO multi-nível com realimentação

- Três Filas*:*
  - *Q0* – *RR* com *time quantum* 8 ms
  - *Q1* – *RR* com *time quantum* 16 ms
  - *Q2* – *FCFS*
- Escalonamento
  - Um novo processo começa em Q0. Se esgota os 8ms antes de bloquear, passa para Q1
  - Em Q1, se o processo ao executar esgota 16ms antes de bloquear passa para Q2

FIFO multi-nível com realimentação



Linux scheduler

- o Linux considera três classes "clássicas" de *scheduling*, cada uma incorporando prioridades múltiplas, que, quando ordenadas por ordem decrescente de prioridade, são:
  - *SCHED_FIFO* – classe formada por processos cuja atribuição do processador só lhes é retirada quando processos da mesma classe, com prioridade mais alta, estão prontos a serem executados (*priority superseded*);
  - *SCHED_RR* – classe formada por processos cuja atribuição do processador esta condicionada a uma janela de execução, a atribuição do processador é-lhes retirada mais cedo quando processos da classe *SCHED_FIFO*, ou da mesma classe com prioridade mais alta, estão prontos a serem executados (*priority superseded*);
  - *SCHED_OTHER* – classe formada pelos processos restantes, o processador só é atribuído a processos desta classe se não houver outro tipo de processos prontos a serem executados;
- as classes *SCHED_FIFO* e *SCHED_RR* estão associadas a processamento de tempo real e a processos de sistema e o valor das suas prioridades é fixo;
- a classe *SCHED_OTHER* está associada aos processos utilizador;

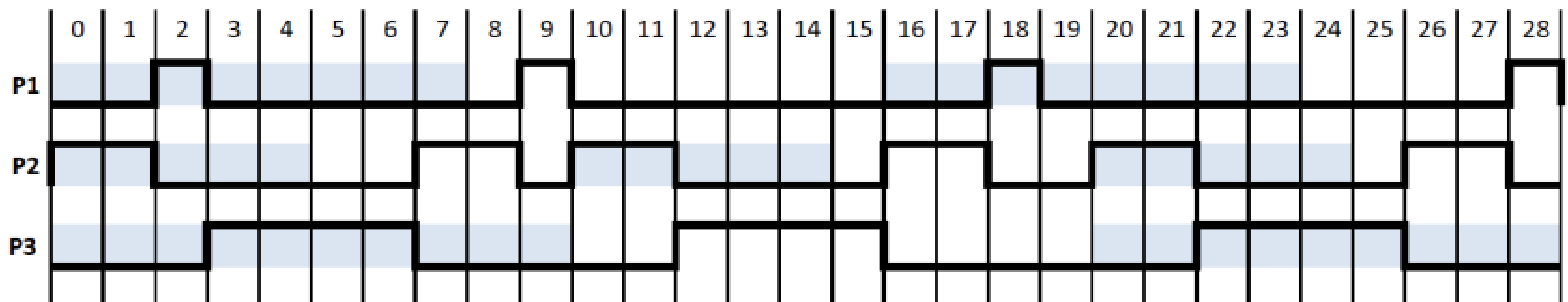
Linux scheduler

- Foram recentemente incorporadas no Linux novas classes de scheduling:
  - *SCHED_DEADLINE* – classe formada por *threads* de tempo real; para cada thread são indicados: período, deadline relativa e tempo de computação; escalonador usa algoritmo *Global Earliest Deadline First;* disponível desde kerner 3.14 (Março 2014)
  - *SCHED_BATCH* – escalonador assume que processos nesta classe são cpu-bound; usa *timeslices* maiores; aplica penalty quando processo acorda.
  - *SCHED_IDLE* – classe formada por processos de muito baixa prioridade; o valor nice não tem efeito neste processos

Ealiest Deadline First

- Escolhe para execução sempre o processo que tem a *deadline* mais próxima
- É óptimo do ponto de vista de que se é possível correr um conjunto de processos (com tempo de chegada, tempo de processamento e deadline) de forma a todos cumprirem as deadlines, o EDF cumpre as deadlines

Process	Execution Time	Period
P1	1	8
P2	2	5
P3	4	10



Linux: algoritmo até 2007

- para a classe *SCHED_OTHER*, o Linux usava um algoritmo baseado em *créditos* no estabelecimento da sua prioridade;
- no instante de recreditação *i*, a prioridade de cada processo (equivalente ao numero de créditos de execução que lhe são atribuídos) e calculada pela fórmula seguinte:

$$CPU_j(i) = \frac{CPU_j(i-1)}{2} + PBase_j + nice_j$$

- em que $CPU_j(i)$ representa a prioridade do processo *j* (o número de créditos que lhe são atribuídos) no instante de recreditação *i*, $CPU_j(i\text{-}1)$ o numero de créditos não usados pelo processo *j* no intervalo de recreditacao *i*-1, $PBase_j$ a prioridade base do processo j e $nice_j$ o valor de alteracão de prioridade dependente do utilizador (valor no intervalo -20 a 19);

Linux: algoritmo até 2007

ieeta universidade de aveiro ua

- o *scheduler* calendariza para execução o processo com mais créditos (maior prioridade); sempre que ocorre uma interrupção do *RTC* o processo perde um crédito; quando o número de créditos atinge o valor zero, o processo perde a posse do processador por esgotamento da janela de execução e outro processo é calendarizado;
- quando já não há processos na fila de espera dos *processos prontos a serem executados* com créditos não nulos, procede-se a uma nova operação de recreditação que envolve todos os processos da classe, mesmo aqueles que estão bloqueados;
- o algoritmo de *scheduling* combina, assim, dois factores, a história passada de execução do processo e a sua prioridade, e maximiza o tempo de resposta dos processos *I/O*-intensivos sem produzir *adiamento indefinido* para os processos *CPU*-intensivos.

Linux: algoritmo até 2007

ieeta universidade de aveiro ua

- Forças
  - Funciona!
  - Algoritmo (relativamente) simples
- Fraquezas
  - Escalabilidade
    - Executa em O(n)
  - Timeslice médio grande
    - 210ms
  - Prioridade a processos intensivos em I/O
  - Comportamento RealTime
    - Kernel não é *preemtable*

Linux: novo algoritmo

- a partir da versao 2.6.23 do *kernel*, o Linux passou a usar um algoritmo de scheduling para a classe *SCHED_OTHER* conhecido pelo nome de *justiça total* (*total fairness*);
- assuma-se um processador ideal que tem tantos elementos de processamento quantos os processos que correntemente coexistem; assim, se existirem N processos, o processador executa-los-á em paralelo distribuindo a potência de cálculo por todos eles de um modo uniforme (a velocidade de processamento será 1/N da velocidade se existisse um só processo em execução);
- o algoritmo vai procurar modelar este comportamento num processador real;
- são definidas duas variáveis associadas a cada processo:
  - *tempo de execução virtual* – tempo de execução associado ao processo se ele estivesse a ser executado no processador ideal;
  - *tempo de espera* – tempo real que o processo aguarda na fila de espera dos *processos prontos a serem executados* pela atribuição do processador;

Linux: novo algoritmo

ieeta
universidade de aveiro
ua

- o *scheduler* organiza os processos numa fila de espera dos *processos prontos a serem executados* única e calendariza para execução o processo cuja diferença entre o *tempo de espera* e o *tempo de execução virtual* é maior, procurando desta forma minimizar o grau de injustiça existente;
- sempre que um processo é calendarizado para execução, o seu *tempo de espera* é decrementado do valor correspondente à janela de execução, o dos restantes presentes na fila de espera é incrementado do mesmo valor, e todos eles tem o *tempo de execução virtual* incrementado do valor de execução virtual;
- os processos bloqueados mantêm os valores destas variáveis inalterados e não entram naturalmente no esquema de seleção enquanto não forem acordados;
- o algoritmo de *scheduling* usa, pois, como elemento central de decisão, a história passada de execução do processo, não havendo propriamente uma distinção entre os processos *I/O*-intensivos e os processos *CPU*-intensivos.

Linux: novo algoritmo

ieeta
universidade de aveiro
ua

- *Runqueue*
  - 1 por CPU
  - Representa as *runnable tasks* desse CPU
  - Tem 2 *priority queues*
    - *Active* e *Expired*
- *Priority Array*
  - Desempenho O(1)
  - Permite acesso a tarefa com prioridade mais alta
  - Tarefas com mesma prioridade são servidas por ordem

Tópico prático: comando **`nice`**

- Através do comando **`nice`** um utilizador pode baixar a prioridade a um processo
  - Executar **`program`** através de:

```
nice -20 program
```

- O comando **`nice`** não tem grande efeito enquanto o número de processadores for suficiente para a execução de todas as tarefas/processos