Escalonamento no Unix/Linux

Comecemos pelo Unix

- Por razões históricas vamos começar por falar do escalonamento histórico do Unix
- Pedagogicamente é interessante conhecer o núcleo do Unix porque ilustra bastante bem muitos dos conceitos e foi a referência utilizada na explicação do Sistemas Operativos durante anos
- No Linux as implementações são geralmente diferentes, mas mantem a generalidade do modelo computacional do Unix

Unix e Linux
Contexto de execução do processo

Unix : Contexto do Processo

- Em Unix, divido em duas estruturas:
  - A estrutura **`proc`**
    - A informação que tem estar disponível para se poder efetuar o escalonamento e o funcionamento dos *signals*
    - Esta informação não pode deixar de estar em memória RAM, mesmo que o processo na sua totalidade tenha sido guardado em disco (*swap*)
  - A estrutura u (user)
    - Todo o resto do contexto núcleo que só é necessário quando processo está em execução
    - Podia estar em disco quando processo não estava em execução para diminuir a necessidade de memória do núcleo

Otimização : Primeiras versões do Unix conseguiam correr em máquinas com 50kB de RAM!

Unix : Contexto do Processo

Estrutura `proc`:

- Parte do contexto necessária para efetuar as operações de escalonamento

- p_stat – estado do processo
- p_pri – prioridade
- p_sig – sinais enviados ao processo
- p_time – tempo que está em memória
- p_cpu – tempo de utilização
- p_pid – identificador do processo
- p_ppid – identificador do processo pai

estrutura `u`:

- Parte do contexto necessária quando processo está em execução

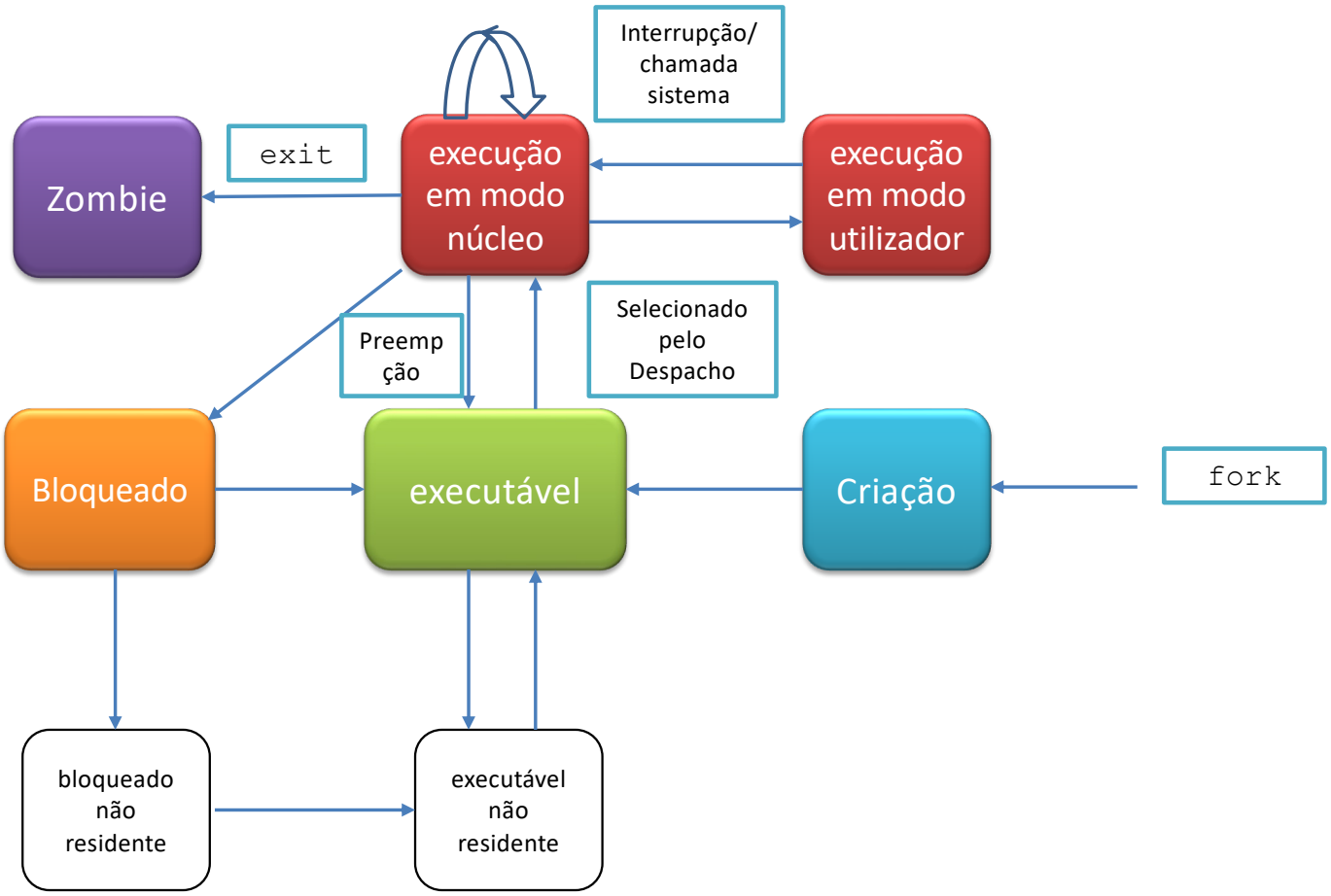
- registos do processador
- pilha do núcleo
- códigos de protecção (UID, GID)
- referência ao directório corrente e por omissão
- tabela de ficheiros abertos
- apontador para a estrutura proc
- parâmetros da função sistema em execução

Linux: contexto do processo

- Em Linux não existe distinção entre tarefas e processos no que diz respeito ao contexto núcleo e ao escalonamento
- Deixou de existir a diferenças da estrutura u e proc que não tinha sentido no hardware atual
- `task_struct` – tem a informação do contexto da tarefa

Header files
- `include/linux/sched.h` – estruturas de dados dos processos
- `include/asm-i386/system.h` – estruturas de dados do contexto de hardware

Diagrama de Estados do Unix



Unix
Escalonamento (Scheduling)

Dois tipos de prioridade

- Cada processo (ou tarefa) tem dois tipos de prioridade, consoante esteja a executar-se em modo utilizador ou núcleo

- **Objetivo:** dar maior prioridade aos processos quando se bloqueiam no núcleo (numa chamada sistema) para que terminem rapidamente e libertem eventuais recursos do núcleo que possam estar a deter, por exemplos *buffers* para leitura/escrita nos discos

Prioridades em modo utilizador

- Prioridade tem uma escala inversa do habitual, valor 0 - maior prioridade, a N - menor prioridade
- As prioridades dos processos em modo utilizador são dinamicamente calculadas em função do tempo de processador utilizado
- Escalonamento é preemptivo em modo utilizador.
  - Quando o processo retorna de modo núcleo é verificado se existe outro mais prioritário, efetuando-se a comutação

Algoritmo de escalonamento

- O escalonador escolhe o processo mais prioritario durante um *time slice* (100 ms, valor de compromisso)
- *Round-robin* entre os que têm a prioridade mais elevada (multilista habitual)
- Ao fim de 1 segundo (50 *ticks* de 20 ms) , o escalonador acorda e recalcula prioridades dos processos em modo utilizador:
- Para cada processo:
  - TempoProcessador = TempoProcessador / 2
  - Prioridade = TempoProcessador/2 + PrioridadeBase + *nice*

Esquecimento progressivo do uso mais antigo do CPU
o tempo de CPU diminui exponencialmente com o numero de períodos de escalonamento

Escalonamento privilegia os processos interactivos

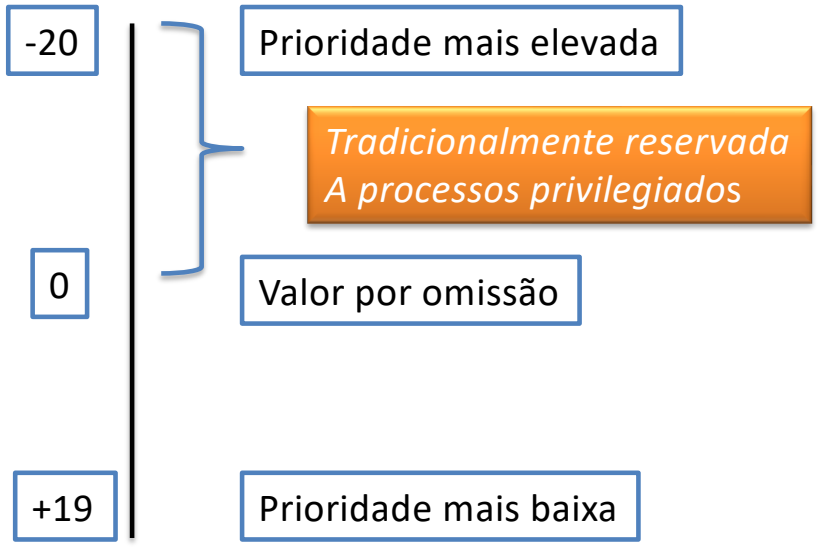
- Este cálculo reduz a prioridade dos processos que tenham usado mais tempo de CPU recentemente, atribuindo-lhes um valor maior e consequentemente uma menor prioridade.

- Os processos E/S intensivos tendem, por um lado, a ser mais prioritários do que os processos CPU intensivos, e, por outro lado, quando se bloqueiam em modo núcleo, é-lhes dada a oportunidade de saírem de modo núcleo rapidamente quando forem desbloqueados.

Nice (be nice to others….)

Ajuste da prioridade pelo utilizadores

- O valor de *nice* tem um valor no intervalo [-20, 19] que é adicionado a prioridade base no calculo do valor da prioridade
- O valor de *nice* por omissão é 0 e pode ser modificado por chamada sistema
- Só com privilégios se podem usar valores negativos (maior prioridade)
- No Linux houve alterações significativas do modo como o *nice* afeta a prioridade



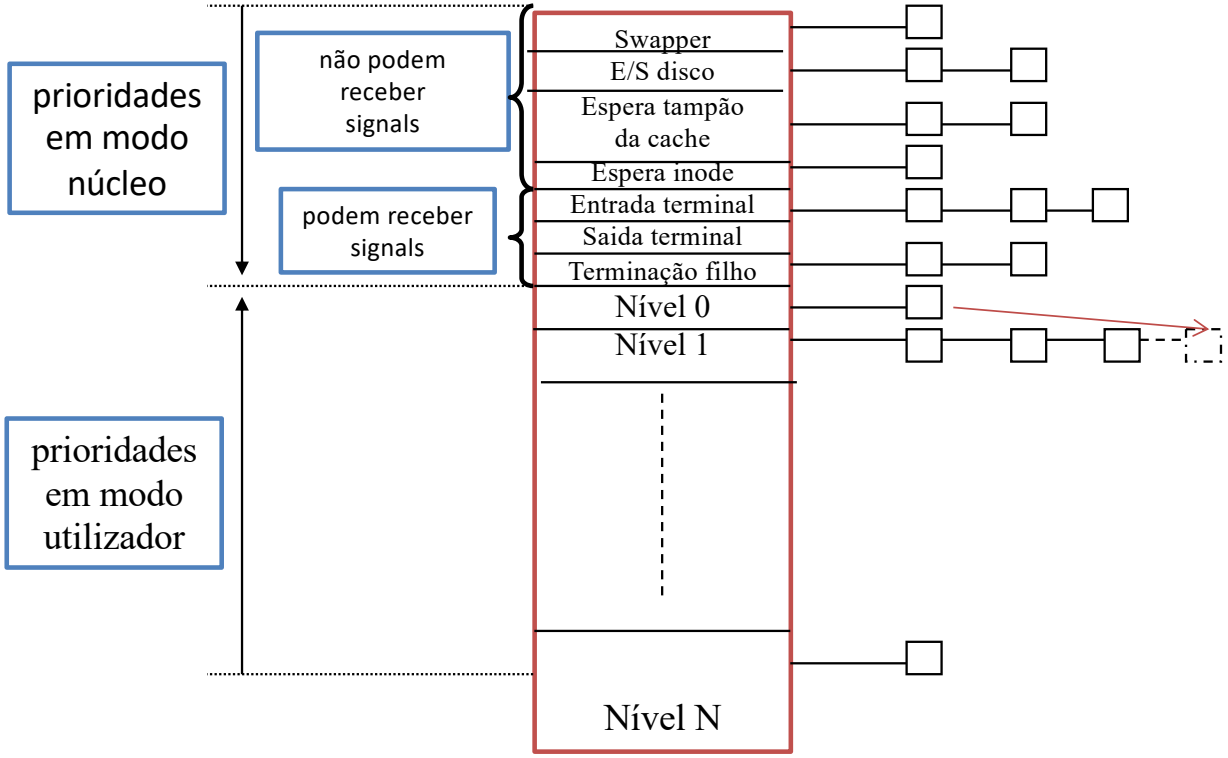
Unix: Prioridades em modo núcleo

- O processo em modo núcleo não é comutado
- Se se bloquear à espera de acontecimento é lhe atribuída uma prioridade fixa definidas com base no acontecimento que o processo está a tratar
- As prioridades em modo núcleo são sempre superiores às prioridades utilizador (quando se tornar executável irá retirar o CPU a qualquer processo que esteja a executar-se em modo utilizador)
- Quando passa de modo núcleo para modo utilizador (retorno da *system call*) é lhe naturalmente atribuída um prioridade utilizador recalculada nesse momento de acordo com a formula habitual

Unix: Prioridades

- Depende do tipo de recursos que o processo detém quando é bloqueado
- Definida quando o processo é bloqueado

- Actualizadas periodicamente
- Depende do tempo de processador já usado



Resumindo...

- Prioridades negativas mais altas (valor abs.)
  - Processos que se bloquearam no núcleo ao acederem a recurso crítico
- Prioridades negativas mais baixas
  - Idem mas recurso menos crítico
- Prioridades positivas mais baixas
  - Processos em modo utilizador que, no passado recente, usaram pouco o CPU
- Prioridades positivas mais altas
  - Processos em modo utilizador que, no passado recente, usaram mais o CPU

Como escala com muitos processos?

- Todos os segundos é necessário efetuar o cálculo das prioridades de todos os processos
- Má escalabilidade com um número elevado de processos
- Funcionava bem para sistemas interativos, mas não para servidores com elevadas cargas

Escalonamento no Linux

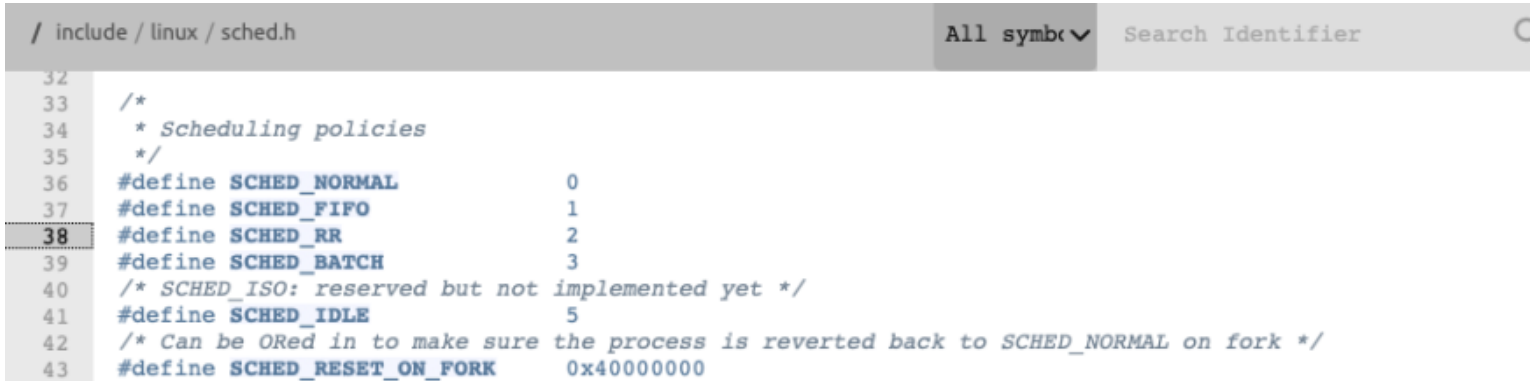
Completely Fair Scheduler

- O escalonamento tem evoluído com diferentes versões do Linux
- A sua comparação e detalhe é interessante, mas transcende o objetivo deste curso
- Vamos considerar o *Completely Fair Scheduler* (CFS) disponível desde a versão 2. 6

- Linux v1.2 – Round Robin
- Linux v2.2 – Scheduling Classes & Policies, Categorizing tasks as
- non/real-time, non-preemptible
- Linux v2.4 – Division in epochs, goodness of function
- Linux v2.6.0 – v2.6.22 –O(1), Runqueues & priority arrays
- Linux v2.6.23 (and after) – Completely Fair Scheduler (CFS)

Classes de escalonamento

- Classe de escalonamento
  - Solução modular para permitir diferentes politicas
  - Cada *task* pertence a uma *scheduling class*
  - Algumas classes:
    - SCHED_NORMAL – Por omissão a politica de escalonamento(CFS)
    - SCHED_RR – escalonamento *round-robin* (tempo real) com *time slice*
    - SCHED_FIFO – Tarefas criticas tempo real sem *time slice*



Completely Fair Scheduler (CFS)

- **Objetivo:** criar um escalonamento que seja próximo de uma multiplexagem equitativa *ideal* por todos os processos
- Num sistema ideal com *N* processos, cada um deveria receber 1/*N* do tempo do CPU
- Se a comutação fosse com intervalos de tempo muito pequeno e durante um período largo seria possível criar esta abstração....
- ...mas a mudança de contexto tem custos: *time-slices* muito pequenos são ineficientes!
- O CFS não tem a noção de *time slice* de duração fixa. Dependendo dos processos executáveis calcula quanto tempo de CPU cada processo executável deveria ter
- O CFS não tem prioridades atribuídas aos processos

Decisão de escalonamento - *vruntime*

- Cada processo tem atributo *vruntime*
  - Representa o tempo acumulado de execução em modo utilizador do processo
  - Quando processo perde CPU, seu *vruntime* é incrementado com o tempo executado nesse *time slice*
  - Processo mais prioritário é o com vruntime mínimo
    - Novo processo recebe *vruntime* igual ao *vruntime* mínimo entre os processos ativos
- Só o valor do *vruntime* é importante para o escalonamento

Porque não 0?

- O *nice* passou a ser uma forma de pesar este fatia de tempo. Processos com *nice* elevado recebem um fatia proporcionalmente menor que processos com *nice* mais baixo

Calculo dinâmico do *time slice (1/3)*

- O CFS visa **aproximar** um escalonador ideal que alterna a execução dos processos com um período **infinitamente pequeno**:
  - Este escalonador ideal garante que o *vrutime* é **sempre** idêntico para todos os processos (caso os processos nunca se bloqueiem)

- A precisão desta aproximação é controlada pela variável *targeted latency:*
  - Neste período cada tarefa executável deveria correr pelo menos uma vez
  - A diferença máxima entre vruntimes é no máximo *targeted latency:*
  - *targeted latency* define o erro máximo que se deseja tolerar nesta aproximação de um escalonador ideal

Calculo dinâmico do *time slice (2/3)*

- Vamos supor um *targeted latency* de 20 ms (valor por omissão em Linux)
  - com 4 processos, teríamos um tempo atribuído a cada um de 5ms

- O problema advém se tivermos 200 processos executáveis
  - o tempo atribuível a cada processo seria de apenas 0,1ms
  - **penalização muito grande pela comutação de processos**

Calculo dinâmico do *time slice (3/3*

- Este problema é resolvido introduzindo um segundo conceito:
  - *minimum granularity*, período em que a tarefa não será *preempted*, excepto se se bloquear
  - Se *minimum granularity*=1 ms e houver 200 processos:
    - 200 * 1 ms = 200 ms
    - a *targeted latency* de 20ms já não é atingível

    ➔ erro maior em aproximar um escalanador ideal *"completely fair"*

- A latência de escalonamento efetiva torna-se:

  *sched_latency = max (num_exec_tasks *min_granularity, targeted_latency)*

- O sistema é equilibrado para cargas típicas de até uma dezena de processos executáveis

Nice no CFS

- O *nice* serve para pesar a distribuição do tempo.
- Em vez de ser dividido por todas as tarefas é divido pelo somatório dos respetivos *nices,*
- Os *nices* do intervalo [-20,19] são mapeados num valor de acordo com a tabela

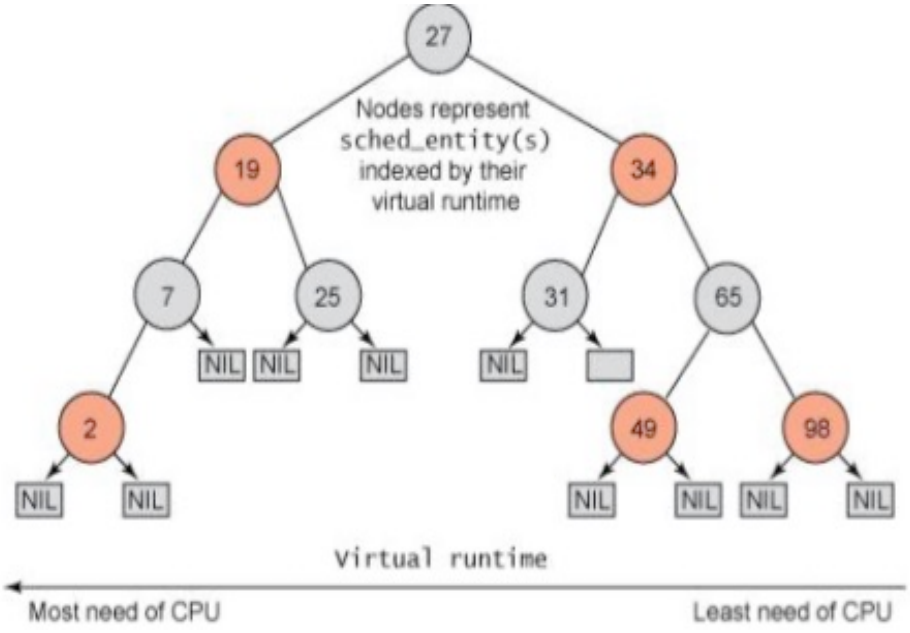
```
static const int prio_to_weight[40] = {
 /* -20 */     88761,     71755,     56483,     46273,     36291,
 /* -15 */     29154,     23254,     18705,     14949,     11916,
 /* -10 */      9548,      7620,      6100,      4904,      3906,
 /*  -5 */      3121,      2501,      1991,      1586,      1277,
 /*   0 */      1024,       820,       655,       526,       423,
 /*   5 */       335,       272,       215,       172,       137,
 /*  10 */       110,        87,        70,        56,        45,
 /*  15 */        36,        29,        23,        18,        15,
};
```

$$\texttt{time\_slice}_k = \frac{\texttt{weight}_k}{\sum_{n=0}^{n-1} \texttt{weight}_i} \cdot \texttt{sched\_latency}$$

Completely Fair Scheduler (CFS)

- Processos executáveis mantidos em *red-black tree* ordenada por *vruntime*:
  - complexidade assimptótica de inserir uma tarefa desce de *O(n)* para *O(log(n))*, onde n é o número de tarefas
  - Otimização (*O(1))*para ter sempre a referência da tarefa com menor valor sem percorrer a arvore (nó mais à esquerda na árvore)

- *Self-balancing binary search tree*



Conceito no CFS

- Afinidade de uma tarefa a um processador.
  - Evitar se possível que a tarefa seja executada noutro processador conduzindo a invalidação de caches
- Filas de escalonamento: uma para cada processador
- Balanceamento de carga
  - Quando se torna necessário reequilibrar o sistema porque os mecanismos normais de afinidade tendem a ter alguns processadores muito pouco carregados

Conclusões

No Unix e Linux o escalonamento é basicamente em *round robin time-sharing,* mas tem evoluído para se adaptar a cargas de tarefas muito elevadas de servidores com múltiplos CPU.

Apesar de soluções que duram décadas é um tema ainda sujeito a muito trabalho de investigação.

O CFS tem se mantido com diversas otimizações nos sistemas atuais