Escalonamento

- Objetivo
  - Entender o papel do escalonamento e da análise de escalonabilidade na previsão de que aplicações de tempo real cumprem seus deadlines
- Tópicos
  - Modelo de processos simples
  - A abordagem executivo cíclico
  - Escalonamento baseado em processos
  - Testes de escalonabilidade baseados em utilização
  - Análise de tempo de resposta para PF e EDF
  - Tempo de execução no pior caso
  - Processos esporádicos e aperiódicos
  - Sistemas de processos com D < T
  - Interação entre processos, bloqueio e protocolos de teto de prioridade
  - Um modelo de processos extensível
  - Sistemas dinâmicos e análise on-line
  - Programação de sistemas baseados em prioridade

Escalonamento

- Em geral, um esquema de escalonamento provê duas propriedades:
  - Um algoritmo para ordenar o uso dos recursos do sistema (especialmente o processador)
  - Um meio de prever o comportamento no pior caso do sistema quando o algoritmo de escalonamento é aplicado
- A previsão pode então ser usada para confirmar os requisitos temporais da aplicação

Modelo de Processos Simples

- A aplicação é suposta consistir de um conjunto fixo de processos
- Todos os processos são periódicos, com períodos conhecidos
- Os processos são completamente independentes um do outro
- Todos os overheads do sistema, chaveamento de contexto, etc, são ignorados (é assumido custo zero)
- Todos os processos tem um deadline igual ao seu período (ou seja, cada processo deve estar concluído antes de sua próxima liberação)
- Todos os processos possuem um tempo de execução no pior caso fixo (WCET)

Notação Convencional

`B`	Tempo de bloqueio do processo no pior caso (se aplicável)
`C`	Tempo de computação no pior caso (WCET) do processo
`D`	Deadline do processo
`I`	O tempo de interferência que o processo sofre
`J`	Release jitter do processo
`N`	Número de processos no sistema
`P`	Prioridade atribuída ao processo (se aplicável)
`R`	Tempo de resposta no pior caso do processo
`T`	Tempo mínimo entre duas liberações do processo (período)
`U`	A utilização de cada processo (igual a C/T)
`a-z`	O nome do processo

Executivo Cíclico

- Uma maneira comum de implementar sistemas de tempo real críticos é usar um executivo cíclico
- Aqui o projeto é concorrente, mas o código é produzido como uma coleção de procedimentos
- Procedimentos são mapeados sobre um conjunto de ciclos menores (minor cycles) que formam juntos a escala completa ou ciclo maior (major cycle)
- O Minor cycle determina o mínimo ciclo de tempo
- O Major cycle determina o máximo tempo de ciclo

Tem a vantagem de ser completamente determinístico

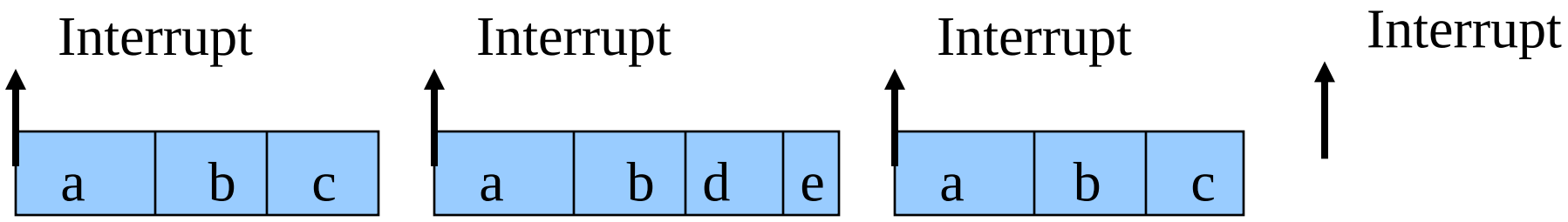
Considere o conjunto de processos

Processo	Período, T	Tempo de computação, C
a	25	10
b	25	8
c	50	5
d	50	4
e	100	2

Executivo Cíclico

```
loop
  wait_for_interrupt;
  procedure_for_a; procedure_for_b; procedure_for_c;
  wait_for_interrupt;
  procedure_for_a; procedure_for_b; procedure_for_d;
  procedure_for_e;
  wait_for_interrupt;
  procedure_for_a; procedure_for_b; procedure_for_c;
  wait_for_interrupt;
  procedure_for_a; procedure_for_b; procedure_for_d;
end loop;
```

Linha de tempo do conjunto de processos



Propriedades

- Nenhum processo realmente existe em tempo de execução; cada minor cycle é apenas uma seqüência de chamadas de procedimentos
- Os procedimentos compartilham um espaço de endereçamento comum, e podem então passar dados entre eles. Esses dados não precisam ser protegidos (via um semáforo, por exemplo) porque o acesso concorrente não é possível
- Todos os períodos dos "*processos*" devem ser múltiplos do tempo de ciclo menor

Problemas com Executivos Cíclicos

- Dificuldade de incorporar processos com períodos longos; o tempo de major cycle é o máximo período que pode ser acomodado sem escalas secundárias
- Atividades esporádicas são difíceis (as vezes impossíveis) de serem incorporadas
- O executivo cíclico é difícil de construir e manter — é um problema NP-hard
- Qualquer “processo” com tempo de computação maior precisará ser dividido em um número fixo de procedimentos com tamanho mediano (isto pode dilacerar a estrutura do código em uma perspectiva de engenharia de software, sendo portanto muito sujeito a bugs)
- Métodos mais flexíveis de escalonamento são difíceis de suportar
- Determinismo não é necessário, mas previsibilidade o é

Escalonamento baseado em processos

- Abordagens de escalonamento

  - Fixed-Priority Scheduling (FPS)
  - Earliest Deadline First (EDF)
  - Value-Based Scheduling (VBS)

Fixed-Priority Scheduling (FPS)

- Esta é a abordagem mais amplamente utilizada e é o foco principal deste curso
- Cada processo possui uma prioridade fixa, estática, a qual é definida antes da execução
- Os processos aptos são executados na ordem determinada pelas duas prioridades
- Em sistemas de tempo real, a “prioridade” de um processo deriva de seus requisitos temporais, e não de sua importância para o correto funcionamento do sistema ou para sua integridade

Earliest Deadline First (EDF) Scheduling

- Os processos aptos são executados na ordem determinada pelos deadlines absolutos dos processos
- O próximo processo a executar é aquele com o menor (mais próximo) deadline
- Apesar de ser usual saber os deadlines relativos de cada processo (por exemplo, 25ms após a chegada), os deadlines absolutos são computados em tempo de execução e desta forma o esquema é descrito como dinâmico

Value-Based Scheduling (VBS)

- Se um sistema pode entrar em sobrecarga, então o uso de simples prioridades ou deadlines estáticos não é mais suficiente; um esquema mais adaptativo é necessário
- Isto freqüentemente toma a forma de atribuir um valor tpara cada processo e empregar em tempo de execução um algoritmo de escalonamento baseado em valor para decidir qual processo executa a seguir

Preempção e Não Preempção

- Com esclonamento baseado em prioridade, um processo de alta prioridade pode ser liberado durante a execução de um com baixa prioridade
- Em um esquema preemptivo, haverá um imediato chaveamento para o processo de alta prioridade
- Em um esquema não preemptivo, o processo de baixa prioridade poderá concluir sua execução antes que o outro execute
- Esquemas preemptivos permitem aos processos de mais alta prioridade serem mais reativos (responsivos), e portanto são preferidos
- Estratégias alternativas permitem um processo de baixa prioridade continuar sua execução por algum tempo
- Esses esquemas são conhecidos como deferred preemption ou cooperative dispatching
- Esquemas tais como EDF e VBS podem também tomar a forma preemptiva ou não preemptiva

FPS e Rate Monotonic

- Cada processo recebe uma (única) prioridade baseada em seu período; quanto menor o período, maior a prioridade
- Para dois processos `i` e `j`, $T_i < T_j \Rightarrow P_i > P_j$

- Esta atribuição é ótima no sentido de que se qualquer conjunto de processos pode ser escalonado (usando escalonamento preemptivo baseado em prioridade) com um esquema de atribuição de prioridades fixas, então este dado conjunto de processos também será escalonável com o esquema de atribuição rate monotonic
- Atenção, prioridade 1 é a mais baixa (menor) prioridade

Exemplo de Atribuição de Prioridades

Processo	Período, T	Prioridade, P
a	25	5
b	60	3
c	42	4
d	105	1
e	75	2

Análise Baseada em Utilização

- Apenas para conjuntos de tarefas com D=T
- Existe um teste de escalonabilidade simples, suficiente mas não necessário

$$U \equiv \sum_{i=1}^{N} \frac{C_i}{T_i} \leq N\,(2^{1/N} - 1)$$

$$U \leq 0.69 \text{ as } N \to \infty$$

Limites de Utilização

N	Limite de Utilização
1	100.0%
2	82.8%
3	78.0%
4	75.7%
5	74.3%
10	71.8%

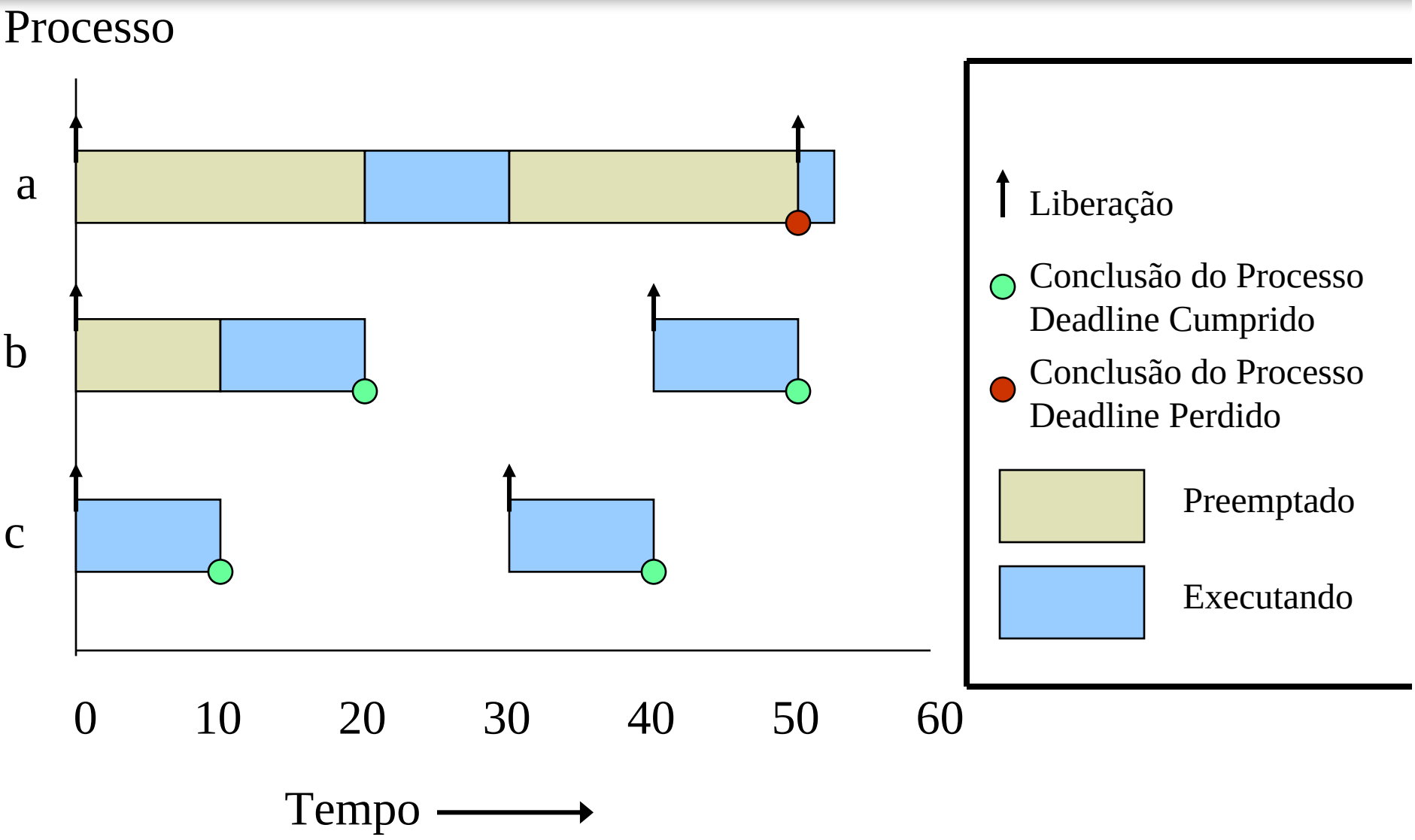
Aproxima-se de 69.3% asimtoticamente

Conjunto de Processos A

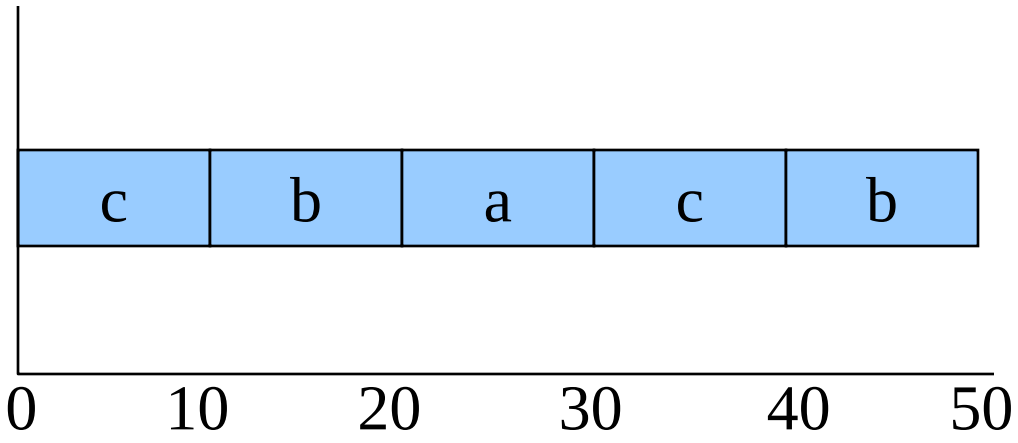
Processo	Período T	WCET C	Prioridade P	Utilização U
a	50	12	1	0.24
b	40	10	2	0.25
c	30	10	3	0.33

- A utilização combinada é 0.82 (ou 82%)
- Isto está acima do limiar para três processos (0.78) e, então, este conjunto de processos é reprovado no teste de utilização

Linha de Tempo:Conjunto de Processos A



Gantt Chart para o Conjunto A



Conjunto de Processos B

Processo	Período T	WCET C	Prioridade P	Utilização U
a	80	32	1	0.400
b	40	5	2	0.125
c	16	4	3	0.250

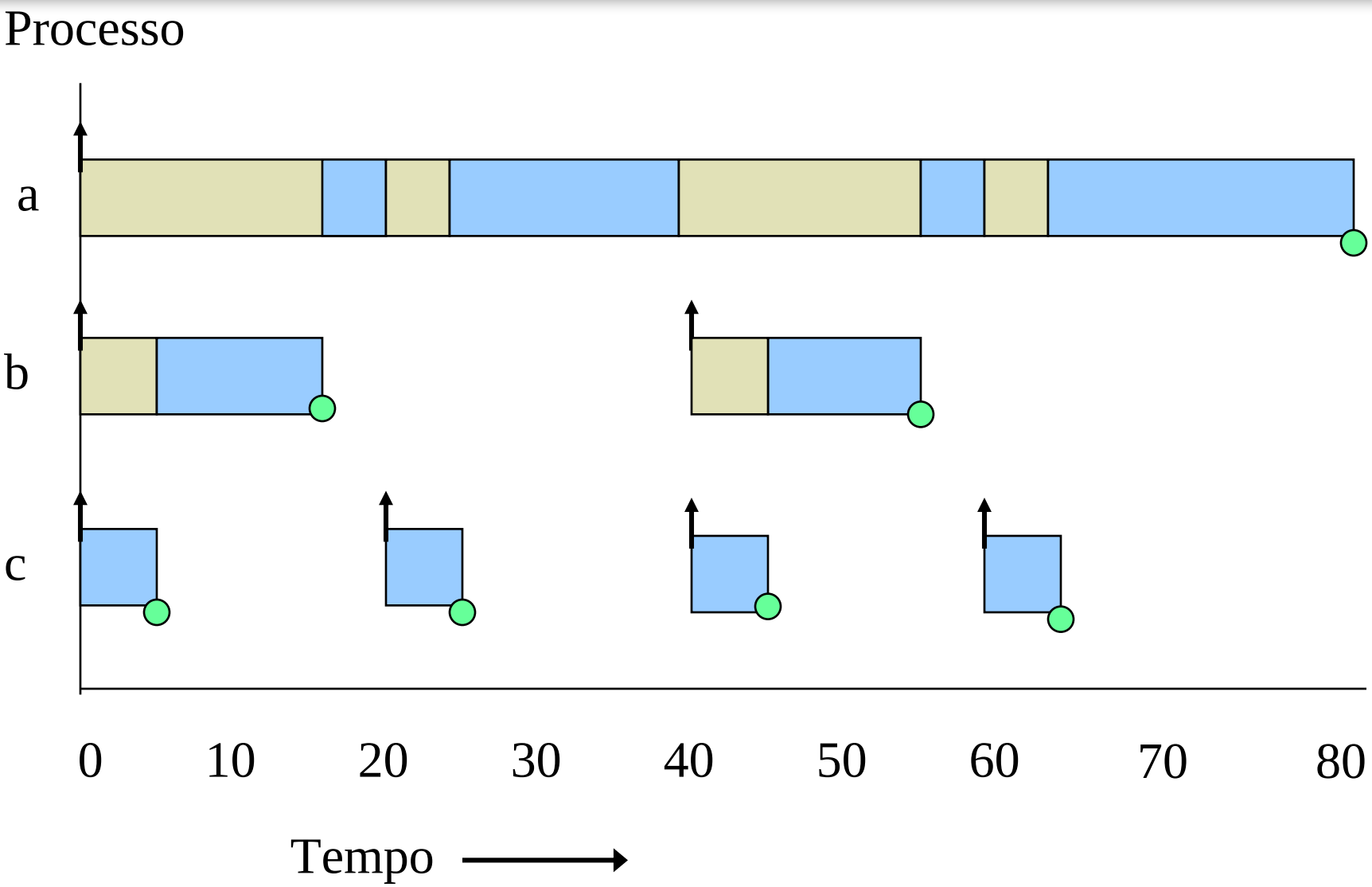
- A utilização combinada é 0.775
- Isto está abaixo do limiar para três processos (0.78) e, portanto, este conjunto de processos irá cumprir todos os seus deadlines

Conjunto de Processos C

Processo	Período T	WCET C	Prioridade P	Utilização U
a	80	40	1	0.50
b	40	10	2	0.25
c	20	5	3	0.25

- A utiilização combinada é 1.0
- Isto está acima do limiar para três processos (0.78) mas o conjunto de processos irá cumprir todos os seus deadlines

Linha de Tempo:Conjunto de Processos C



Crítica aos Testes Baseados em Utilização

- Não são exatos
- Não são gerais
- MAS o teste é O(N)

O teste é dito ser suficiente mas não necessário

Teste Baseado em Utilização para EDF

$$\sum_{i=1}^{N} \frac{C_i}{T_i} \leq 1$$

Um teste muito mais simples

- Melhor que FPS, pode suportar utilizações mais altas. Porém:
- FPS é mais fácil de implementar, as prioridades são estáticas
- EDF é dinâmico e requer um sistema em tempo de execução mais complexo o qual apresenta maior overhead
- É mais fácil incorporar processos sem deadlines em FPS; dar ao processo um deadline arbitrário é mais artificial
- É mais fácil incorporar outros fatores na noção de prioridade do que incorporar na noção de deadline
- Durante situações de sobrecarga (overload)
  - FPS é mais previsível; Processo baixa prioridade perde deadline antes
  - EDF é imprevisível; efeito dominó pode ocorrer no qual um grande número de processos perde deadlines

Análise do Tempo de Resposta

- O tempo de resposta no pior caso da tarefa i é denotado por R, ele é calculado e então comparado (trivialmente) com o deadline da tarefa

$$R_i \leq D_i$$

$$R_i = C_i + I_i$$

Onde I é a interferência recebida das tarefas com prioridade mais alta

Calculando R

Durante *R*, cada tarefa mais prioritária *j* executará um número de vezes:

$$\text{Número de Liberações} = \left\lceil \frac{R_i}{T_j} \right\rceil$$

A função ceiling $\lceil \; \rceil$ fornece o menor inteiro que seja maior que o número fracional que é o seu parâmetro. Assim o ceiling de 1/3 é 1, de 6/5 é 2, e de 6/3 é 2.

Interferência total é dada por:

$$\left\lceil \frac{R_i}{T_j} \right\rceil C_j$$

Equação do Tempo de Resposta

$$R_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$$

onde *hp*(*i*) é o conjunto de tarefas com prioridade mais alta do que a tarefa *i*

Resolvida pela formação de uma relação de recorrência:

$$w_i^{n+1} = C_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n}{T_j} \right\rceil C_j$$

O conjunto de valores $w_i^0, w_i^1, w_i^2, ..., w_i^n, ..$ é monotonicamente não decrescente

Quando $w_i^n = w_i^{n+1}$ a solução para a equação foi encontrada, w_i^0 não pode ser maior do que R_i (usar 0 or C_i)

Cálculo do Tempo de Resposta

```
for i in 1..N loop – para cada processo
  n := 0
  w_i^n := C_i
loop
    calculate new w_i^(n+1)
    if  w_i^(n+1) = w_i^n then
       R_i = w_i^n
      exit value found
    end if
    if  w_i^(n+1) > T_i  then
      exit value not found
    end if
    n := n + 1
  end loop
end loop
```

Process Set D

Processo	Período T	WCET C	Prioridade P
a	7	3	3
b	12	3	2
c	20	5	1

$$R_a = 3$$

$$w_b^0 = 3$$

$$w_b^1 = 3 + \left\lceil \frac{3}{7} \right\rceil 3 = 6$$

$$w_b^2 = 3 + \left\lceil \frac{6}{7} \right\rceil 3 = 6$$

$$R_b = 6$$

$$w_c^0 = 5$$

$$w_c^1 = 5 + \left\lceil \frac{5}{7} \right\rceil 3 + \left\lceil \frac{5}{12} \right\rceil 3 = 11$$

$$w_c^2 = 5 + \left\lceil \frac{11}{7} \right\rceil 3 + \left\lceil \frac{11}{12} \right\rceil 3 = 14$$

$$w_c^3 = 5 + \left\lceil \frac{14}{7} \right\rceil 3 + \left\lceil \frac{14}{12} \right\rceil 3 = 17$$

$$w_c^4 = 5 + \left\lceil \frac{17}{7} \right\rceil 3 + \left\lceil \frac{17}{12} \right\rceil 3 = 20$$

$$w_c^5 = 5 + \left\lceil \frac{20}{7} \right\rceil 3 + \left\lceil \frac{20}{12} \right\rceil 3 = 20$$

$$R_c = 20$$

Revisitando: Conjunto de Processos C

Processo	Período T	WCET C	Prioridade P	Tempo de Resposta R
a	80	40	1	80
b	40	10	2	15
c	20	5	3	5

- A utilização combinada é 1.0
- Isto está acima do limiar de utilização para três processos (0.78), portanto é reprovado no teste
- A análise do tempo de resposta mostra que o conjunto de processos irá cumprir todos os seus deadlines
- RTA é necessário e suficiente

Análise do Tempo de Resposta

- É suficiente e necessária
- Se o conjunto de processos passa no teste eles irão cumprir todos os seus deadlines; Se eles são reprovados no teste então, em tempo de execução, um processo irá perder o seu deadline (a não ser que o tempo de computação estimado sejam muito pessimistas)

Worst-Case Execution Time - WCET

- Obtido por medição ou análise
- O problema com medição é que é difícil estar certo que o pior caso (worst case) foi realmente observado
- A desvantagem da análise é que um modelo apropriado do processador (incluindo caches, pipelines, memory wait states, etc) precisa estar disponível

WCET— Determinando C

Maioria das técnicas de análise envolvem duas atividades distintas

- A primeira pega o processo e decompõe seu código em um grafo dirigido de blocos básicos
- Esses blocos básicos representam seqüências simples de instruções
- O segundo componente da análise pega o código de máquina correspondente a um bloco básico e usa o modelo do processador para estimar o seu tempo de execução no pior caso (worst-case execution time)
- Uma vez que os tempos de todos os blocos básicos são conhecidos, o grafo dirigido pode ser colapsado

Necessária Informação Semântica

```
for I in 1.. 10 loop
   if Cond then
      -- bloco básico com custo 100
   else
      -- bloco básico com custo 10
   end if;
end loop;
```

- Custo simples 10*100 (+overhead), digamos 1005.

- Mas se Cond é verdadeira apenas em 3 ocassiões então o custo é 375

Processos Esporádicos

- Processos esporádicos tem um tempo mínimo entre chegada
- Eles também requerem D<T

- O algoritmo para tempo de resposta no escalonamento de prioridade fixa funciona perfeitamente para valores de D menores que T desde que o critério de parada seja $W_i^{n+1} > D_i$

- Ele também funciona perfeitamente para qualquer ordem de prioridade — hp(i) sempre fornece o conjunto de processos com prioridades mais altas

Processos Hard e Soft

- Em muitas situações os números de pior caso para processos esporádicos são consideravelmente maiores que os números para o caso médio
- Interrupções frequentemente chegam em rajadas (bursts) e uma leitura anormal de sensor pode levar a uma computação adicional significativa
- Avaliar a escalonabilidade com números de pior caso pode levar a uma utilização muito baixa do processador durante a execução do sistema

Regras Gerais

Regra 1 — todos os processos devem ser escalonáveis usando tempo de execução médios e taxas de chegada médias

Regra 2 — todos os processos de tempo real críticos devem ser escalonáveis usando tempos de execução no pior caso e taxas de chegada no pior caso de todos os processos (inclusive os soft)

- Uma conseqüência da Regra 1 é que podem existir situações nas quais não é possível cumprir todos os deadlines correntes
- Esta condição é conhecida como sobrecarga transiente (transient overload)
- Regra 2 garante que nenhum processo hard real-time irá perder o seu deadline
- Se a Regra 2 gerar um nível inaceitável de baixa utilização para "execução normal" então deve-se reduzir o tempo de execução no pior caso (ou a taxa de chegadas)

Processos Aperiódicos

- Esses não possuem um tempo mínimo entre chegadas
- Pode-se executar processos aperiódicos em uma prioridade menor do que as prioridades atribuídas aos processos críticos, logo, em um sistema preemptivo eles não podem tirar recursos dos processos hard
- Mas isto não provê suporte adequado aos processos soft os quais irão freqüentemente perder seus deadlines
- Para melhorar a situação dos processos soft, um servidor pode ser empregado
- Servidores protegem os recursos de processamento necessários para os processos hard mas permitem os processos soft executarem tão cedo quanto possível
- POSIX suporta Sporadic Servers

Conjuntos de Processos com D < T

- Para D = T, Rate Monotonic é ótimo
- Para D < T, Deadline Monotonic é ótimo

$$D_i < D_j \Rightarrow P_i > P_j$$

Exemplo de Conjunto de Processos:D < T

Processo	Período T	Deadline D	WCET C	Prioridade P	Tempo Resposta R
a	20	5	3	4	3
b	15	7	3	3	6
c	10	10	4	2	10
d	20	20	3	1	20

Prova que DMPO é Ótimo

- Deadline monotonic priority ordering (DMPO) é ótimo se qualquer conjunto de processos, Q, que é escalonável pelo esquema de prioridades, W, também é escalonável por DMPO

- A prova da optimalidade do DMPO envolve transformar as prioridades de Q (como atribuídas por W) até que a ordenação seja DMPO
- A cada passo da transformação será preservada a escalonabilidade

Prova do DMPO - Continuação

- Sejam `i` e `j` dois processos (com prioridades adjacentes) em `Q` tais como atribuídas por `W`, $P_i > P_j \wedge D_i > D_j$
- Defina o esquema `W'` como identico a `W` exceto que processos `i` e `j` são trocados

Considere a escalonabilidade de `Q` sob `W'`

- Todos os processos com prioridades maiores que P_i não serão afetados por esta troca em processos de mais baixa prioridade
- Todos os processos com prioridades menores que P_j não serão afetados; Eles irão experimentar a mesma interferência de `i` e `j`
- Processo `j`, o qual era escalonável sob `W`, agora tem uma prioridade maior, sofre menos interferência, e portanto deve ser escalonável sob `W'`

Prova de DMPO – Continuação

- Tudo o que resta mostrar é que o processo i, o qual teve sua prioridade rebaixada, ainda é escalonável
- Sob `W`

$$R_j < D_j, D_j < D_i \; and \; D_i \leq T_i$$

- Logo o processo j apenas interfere uma vez durante a execução de i
- Segue que:

$$R'_i = R_j \leq D_j < D_i$$

- Pode ser concluido que o processo `i` é escalonável após a troca
- Esquema de prioridade `W'` pode ser transformado em `W"` pela escolha de mais dois processos que estão na ordem errada pelo DMPO e troca dos dois

Interações entre Processos e Bloqueio

- Se um processo é suspenso esperando por um processo de mais baixa prioridade completar alguma computação necessária então o modelo de prioridades está, em certo sentido, sendo violado

- É dito que existe uma inversão de prioridades

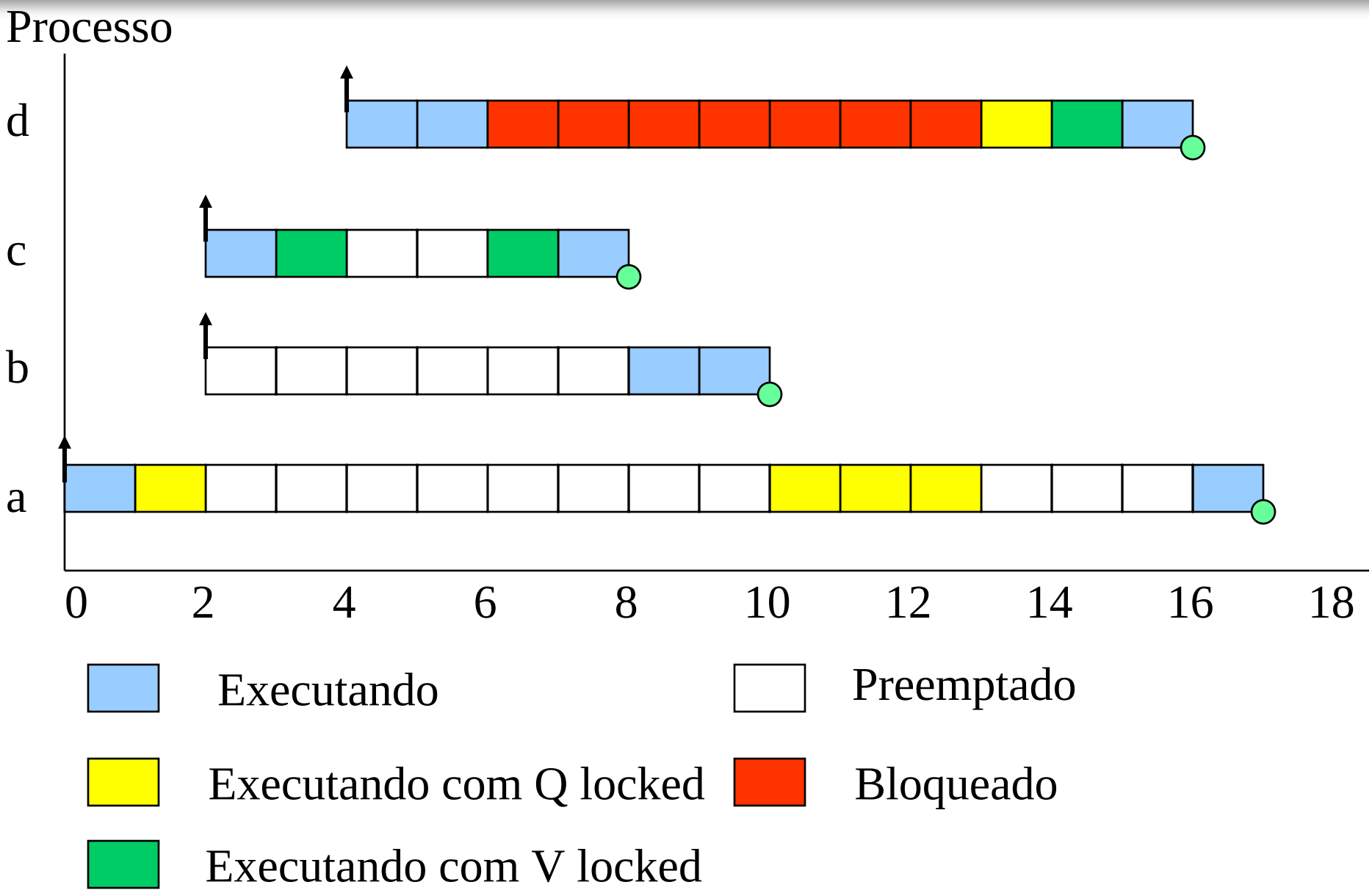
- Se um processo está esperando por outro processo de mais baixa prioridade, é dito que ele está bloqueado

Inversão de Prioridade

- Para ilustrar um exemplo extremo de inversão de prioridades, considere as execuções de quatro processos periódicos: a, b, c e d; e dois recursos: Q e V

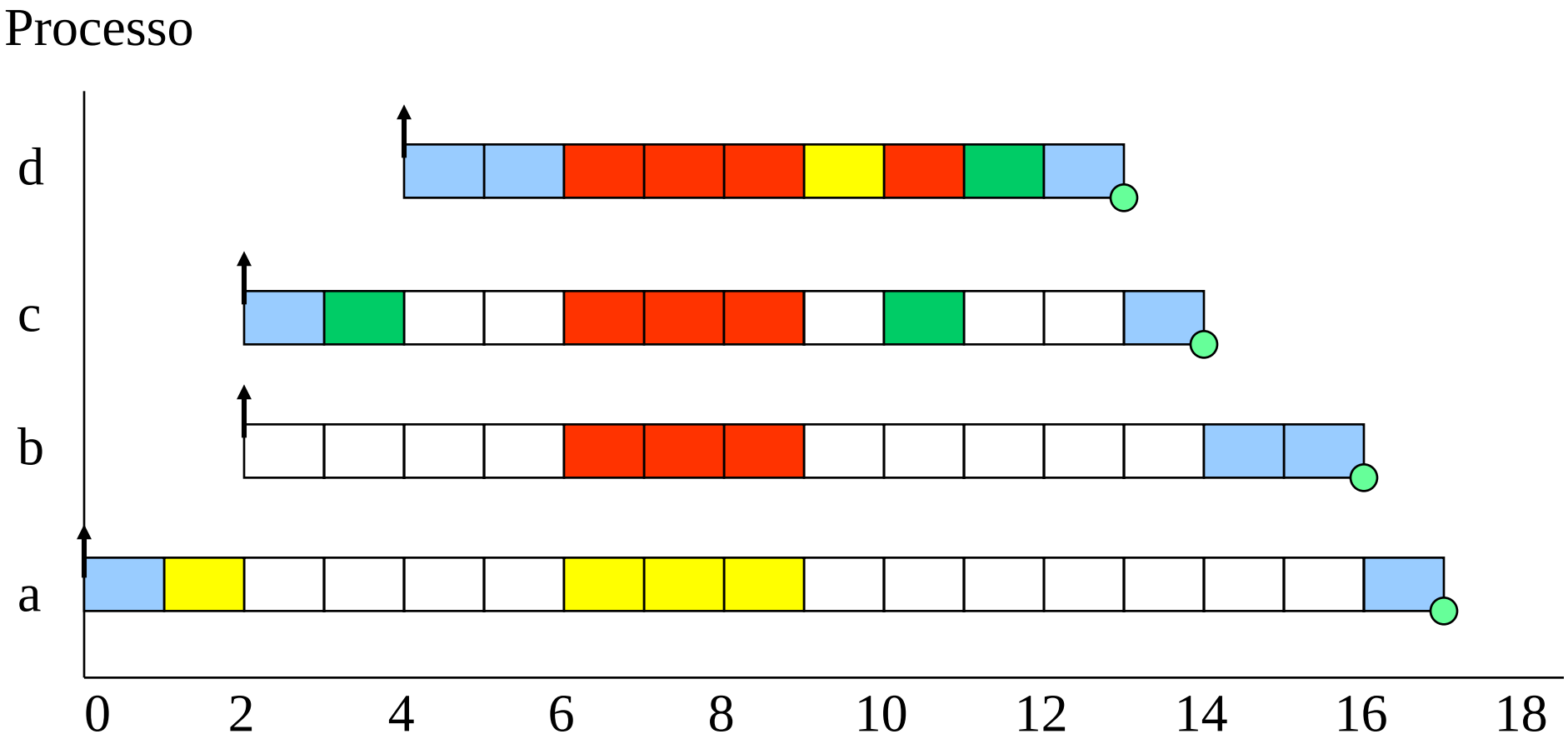
Processo	Prioridade	SeqüênciaExecução	Liberação
a	1	EQQQE	0
b	2	EE	2
c	3	EVVE	2
d	4	EEQVE	4

Exemplo de Inversão de Prioridade



Herança de Prioridade

- Se processo p está bloqueando processo q, então q executa com a prioridade de p



Calculo do Bloqueio

- Se um processo tem `m` seções críticas que podem levar ao seu bloqueio então o máximo número de vezes que ele pode ser bloqueado é `m`
- Se `B` é o tempo máximo de bloqueio e `K` é o número de seções críticas, o processo `i` tem um limite superior para o seu bloqueio dado por:

$$B_i = \sum_{k=1}^{K} usage(k,i)C(k)$$

Tempo de Resposta e Bloqueio

$$R_i = C_i + B_i + I_i$$

$$R_i = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$$

$$w_i^{n+1} = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n}{T_j} \right\rceil C_j$$

Protocolos tipo Priority Ceiling

Duas formas

- Original ceiling priority protocol
- Immediate ceiling priority protocol

Em um Único Processador

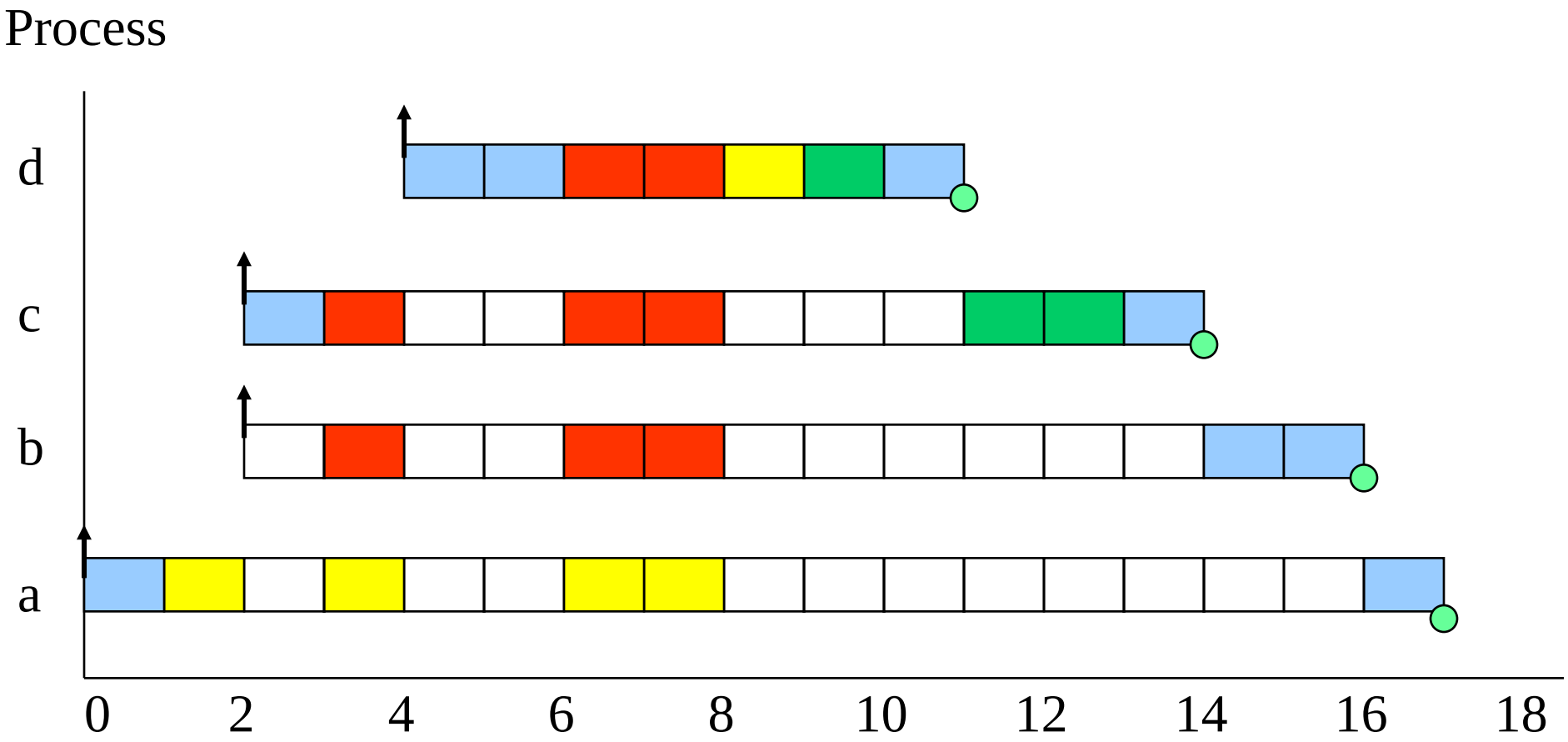
- Um processo de alta prioridade pode ser bloqueado no máximo uma única vez durante a sua execução por processos de mais baixa prioridade
- Deadlocks são evitados
- Bloqueio transitivo é evitado
- Acesso com exclusão mútua a recursos é garantida (pelo próprio protocolo)

OCPP

- Cada processo tem uma prioridade estática default atribuida (talvez pelo esquema deadline monotonic)
- Cada recurso tem um valor de ceiling estático definido, ele é a prioridade máxima dos processos que o utilizam
- Um procersso posui uma prioridade dinâmica que é o máximo entre a sua própria prioridade estática e qualquer prioridade que ele herde devido a estar bloqueando processos mais prioritários
- Um processo somente pode obter um lock em recurso se sua prioridade dinâmica for maior que o ceiling de qualquer recurso atualmente ocupado (excluindo qualquer recurso que o processo já tenha ocupado ele próprio)

$$B_i = \max_{k=1}^{k} usage(k,i)C(k)$$

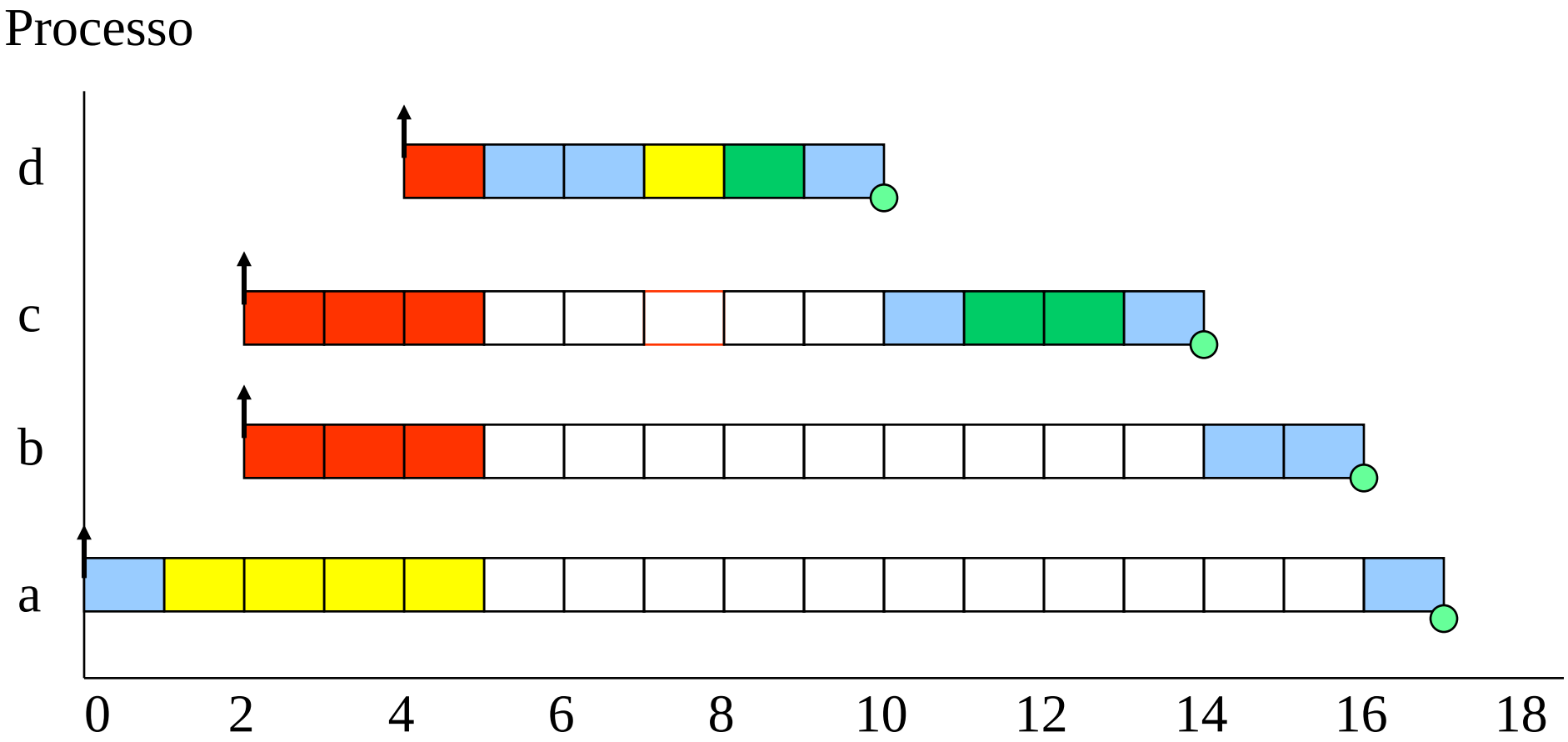
Herança no OCPP



ICPP

- Cada processo possui uma prioridade estática default atribuída (por exemplo, com o esquema deadline monotonic)
- Cada recurso tem um valor de ceiling estático definido, o qual é a prioridade máxima entre os processos que o utilizam
- Um processo tem uma prioridade dinâmica que é o máximo entre sua própria prioridade estática e os valores de ceiling de qualquer recurso que ele tenha ocupado
- Como conseqüência, um processo irá sofrer apenas um bloqueio bem no início de sua execução
- Uma vez que o processo realmente inicia sua execução, todos os recursos que ele necessita estarão livres; Se eles não estivessem, então algum outro processo teria uma prioridade igual ou maior e a execução do processo em questão seria postergada

Herança no ICPP



OCPP versus ICPP

- Apesar do comportamento de pior caso dos dois esquemas de ceiling serem idênticos (de uma perspectiva de escalonamento), existem algumas diferenças:
  - ICCP é mais fácil de implementar que o original (OCPP) pois os relacionamentos de bloqueio não precisam ser monitorados
  - ICPP gera menos chaveamentos de contexto pois o bloqueio é anterior ao início da execução
  - ICPP requer mais movimentos de prioridade pois isto acontece a cada uso de recurso
  - OCPP muda prioridade somente se um bloqueio real acontecer
- Note que ICPP é chamado de Priority Protect Protocol em POSIX e Priority Ceiling Emulation em Real-Time Java

Um Modelo de Processos Extensível

Até agora:

- Deadlines podem ser menores que o período (D<T)
- Processos esporádicos e aperiódicos, assim como processos periódicos, podem ser suportados
- Intereração entre processos é possível, com o tempo de bloqueio resultante sendo incluído nas equações de tempo de resposta

Extensões

- Escalonamento Cooperativo
- Release Jitter
- Deadlines Arbitrários
- Tolerância a Faltas
- Offsets
- Atribuição Ótima de Prioridades

Escalonamento Cooperativo

- Comportamento verdadeiramente preemptivo nem sempre é aceitável em sistemas safety-critical
- Preempção cooperativa ou postergada divide os processos em slots
- Exclusão Mútua é obtida via não preempção
- O uso de preempção postergada tem duas vantagens importantes
  - Ela aumenta a escalonabilidade do sistema, e pode levar a valores menores de C
  - Com preempção postergada, nenhuma interferência pode ocorrer durante o último slot de execução

Escalonamento Cooperativo

- Seja o tempo de execução do bloco final ser F_i

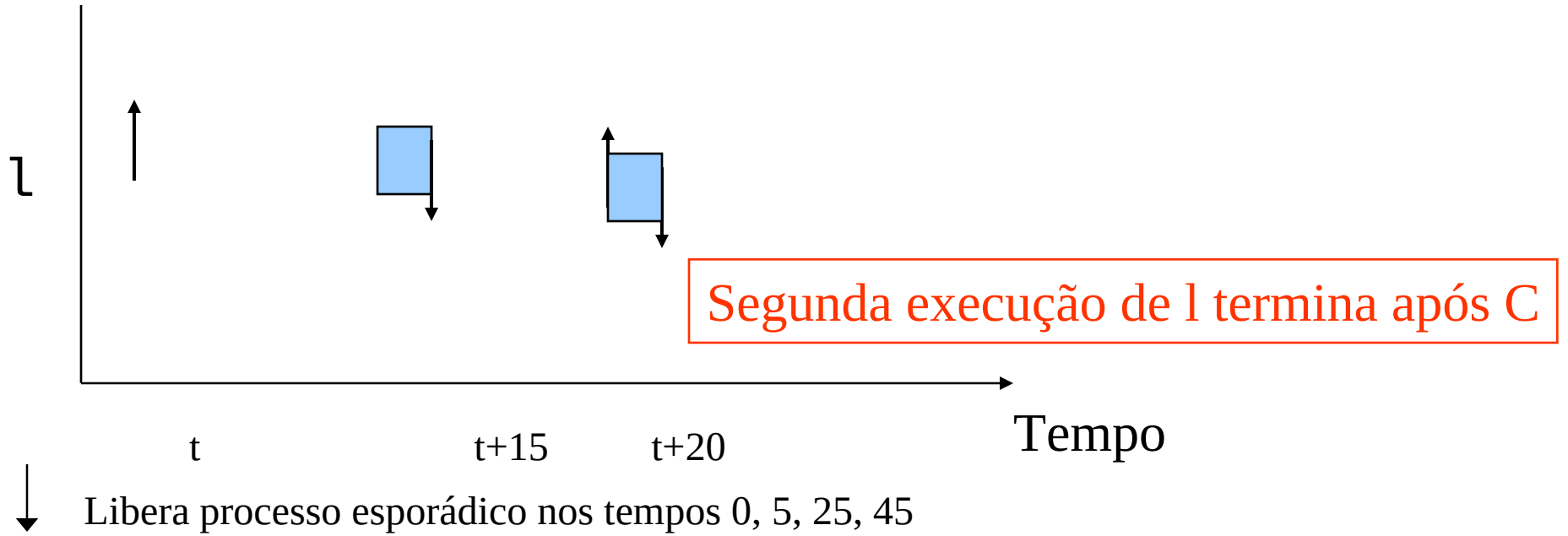
$$w_i^{n+1} = B_{MAX} + C_i - F_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n}{T_j} \right\rceil C_j$$

- Quando isto converge, $w_i^n = w_i^{n+1}$, o tempo de resposta é dado por :

$$R_i = w_i^n + F_i$$

Release Jitter

- Tema central em sistemas distribuídos
- Considere a liberação de um processo esporádico em um processador diferente por um processo periódico, `l`, com um período de 20



Release Jitter

- Esporádica é liberada em `0, T-J, 2T-J, 3T-J`
- Exame da construção da equação de escalonabilidade implica no processo `i` sofrer
  - Uma interferência do processo s se $R_i \in [0, T - J)$
  - Duas interferências se $R_i \in [T - J, 2T - J)$
  - Três interferências se $R_i \in [2T - J, 3T - J)$
- Isto pode ser representado nas equações de tempo de resposta

$$R_i = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i + J_j}{T_j} \right\rceil C_j$$

- Se o tempo de resposta for medido em relação ao real tempo de liberação então o valor de jitter deve ser adicionado

$$R_i^{periodic} = R_i + J_i$$

Deadlines Arbitrários

- Lidam com situações onde D (e possivelmente R) > T

$$w_i^{n+1}(q) = B_i + (q+1)C_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n(q)}{T_j} \right\rceil C_j$$

$$R_i(q) = w_i^n(q) - qT_i$$

- O número de liberações é limitado pelo menor valor de q para o qual a seguinte relação é verdadeira: $R_i(q) \leq T_i$
- O tempo de resposta no pior caso é então o máximo valor encontrado para cada q:

$$R_i = \max_{q=0,1,2,\ldots} R_i(q)$$

Deadlines Arbitrários

- Quando esta formulação é combinada com o efeito do release jitter, duas alterações na análise acima devem ser feitas
- Primeiramente, o fator de interferência deve ser aumentado se qualquer processo de mais alta prioridade sofre release jitter:

$$w_i^{n+1}(q) = B_i + (q+1)C_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n(q) + J_j}{T_j} \right\rceil C_j$$

- A outra alteração envolve o próprio processo. Se ele pode sofrer release jitter então duas janelas consecutivas podem se sobrepor se o tempo de resposta mais jitter é maior que o período.

$$R_i(q) = w_i^n(q) - qT_i + J_i$$

Tolerância a Faltas

- Tolerância a faltas via forward or backward error recovery sempre resulta em computações extras
- Devido a um tratador de exceção ou um bloco de recuperação.
- Em sistemas de tempo real tolerantes a faltas, deadlines ainda precisam ser cumpridos mesmo quando um certo nível de faltas ocorrem
- Este nível de tolerância a faltas é conhecido como o modelo de faltas
- Se o tempo de computação extra resultante de um erro no processo `i` for C_i^f
- $$R_i = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j + \max_{k \in hep(i)} C_k^f$$
- Onde `hep(i)` é o conjunto de processos com prioridade igual ou maior que `i`

Tolerância a Faltas

- Se F é o número de faltas toleradas

$$R_i = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j + \max_{k \in hep(i)} FC_k^f$$

- Se existe um intervalo mínimo entre chegadas T_f

$$R_i = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j + \max_{k \in hep(i)} \left(\left\lceil \frac{R_i}{T_f} \right\rceil C_k^f \right)$$

Offsets

- Até agora foi assumido que todos os processos compartilham um mesmo instante de liberação (instante crítico)

Processo	T	D	C	R
a	8	5	4	4
b	20	10	4	8
c	20	12	4	16

- Com offsets

Processo	T	D	C	O	R
a	8	5	4	0	4
b	20	10	4	0	8
c	20	12	4	10	8

Arbitrary offsets are not amenable to analysis

Análise Não Ótima

- Na maioria dos sistemas reais, os períodos dos processos não são arbitrários mas possivelmente relacionados uns com os outros
- Como no exemplo anterior, dois processos possuem o mesmo período. Nessas situações é fácil atribuir oara um deles um offset (de $T/2$) e analisar o sistema resultante usando uma técnica de transformação que remove o offset — e, portanto, a análise de instante crítico é aplicável.
- No exemplo, processos b e c (tendo o offset de 10) são substituídos por um único processo notacional com período 10, tempo de computação 4, deadline 10 porém nenhum offset

Análise Não Ótima

- Este processo notacional tem 2 importantes propriedades
  - Se ele for escalonável (quando compartilhando um instante crítico com todos os outros processos) então os dois processos reais irão cumprir os seus deadlines quando um recebe o offset (meio período)
  - Se todos os processos de menor prioridade forem escalonáveis quando sofrendo interferência do processo notacional (e de todos os demais processos de alta prioridade) então eles permanecerão escalonáveis quando o processo notacional for substituído pelos dois processos reais (um com o offset)
- Essas propriedades seguem da observação que o processo notacional sempre usa mais (ou o mesmo) tempo de CPU que os dois processos reais

Process	T	D	C	O	R
a	8	5	4	0	4
n	10	10	4	0	8

Parâmetros do Processo Notacional

$$T_n = \frac{T_a}{2} = \frac{T_b}{2}$$

$$C_n = Max(C_a, C_b)$$

$$D_n = Min(D_a, D_b)$$

$$P_n = Max(P_a, P_b)$$

Pode ser extendido para mais que dois processos

Atribuição de Prioridades

Teorema

- Se processo p recebe a menor prioridade e é viável então, se uma ordenação viável de prioridades existe para todo o conjunto de processos, uma ordenação existe com processo p recebendo a prioridade mais baixa

```
procedure Assign_Pri (Set : in out Process_Set; N : Natural;
                      Ok : out Boolean) is
begin
  for K in 1..N loop
    for Next in K..N loop
      Swap(Set, K, Next);
      Process_Test(Set, K, Ok);
      exit when Ok;
    end loop;
    exit when not Ok;  -- falha em encontrar processo viável
  end loop;
end Assign_Pri;
```

Sistemas Dinâmicos e Análise On-line

- Existem aplicações soft real-time dinâmicas nas quais padrões de chegada e tempos de computação não são conhecidos *a priori*
- Apesar de algum nível de análise off-line ser possível, ela não pode mais ser completa e alguma forma de análise on-line é necessária
- A principal tarefa de um esquema de escalonamento on-line é gerenciar qualquer sobrecarga que possa ocorrer devido a dinâmica do ambiente do sistema
- EDF é um esquema de escalonamento dinâmico ótimo
- Durante sobrecargas transientes EDF comporta-se muito mal. É possível acontecer um efeito cascata no qual cada processo perde seu deadline mas utiliza recursos suficientes para fazer o próximo processo perder também

Esquemas de Admissão

- Para conter o efeito dominó, muitos esquemas on-line possuem dois mecanismos:
  - Um módulo de controle de admissão limita o número de processos que são permitidos competir pelo processador, e
  - Uma rotina de despacho baseada em EDF para aqueles processos que são admitidos
- Um algoritmo ideal de admissão previne que os processadores fiquem sobrecarregados de tal forma que a rotina EDF funciona bem

Valor

- Se alguns processos são admitidos, enquanto outros são rejeitados, a importância relativa de cada processo precisa ser conhecida
- Isto é usualmente alcançado através da atribuição de um valor
- Valores podem ser classificados
  - Estático: o processo sempre tem o mesmo valor quando é liberado.
  - Dinâmico: o valor do processo pode somente ser calculado no momento que o processo é liberado (porque ele é dependente de fatores ambientais ou do estado corrente do sistema)
  - Adaptativo: a natureza dinâmica do sistema é tal que o valor do processo vai mudar durante sua execução
- Para atribuir valores estáticos é necessário especialistas no domínio da aplicação articularem seu entendimento sobre o comportamento desejável do sistema

Programming Priority-Based Systems

- Ada
- POSIX
- Real-Time Java

Ada: Real-Time Annex

- Ada 95 has a flexible model:
  - base and active priorities
  - priority ceiling locking
  - various dispatching policies using active priority
  - dynamic priorities

```
subtype Any_Priority is Integer
        range Implementation-Defined;
subtype Priority is Any_Priority range
        Any_Priority'First .. Implementation-Defined;
subtype Interrupt_Priority is Any_Priority range
        Priority'Last + 1 .. Any_Priority'Last;
Default_Priority : constant Priority :=
        (Priority'First + Priority'Last)/2;
```

An implementation must support a range of `Priority` of at least 30 and at least one distinct `Interrupt_Priority`

Assigning Base Priorities

- Using a pragma

```
task Controller is
  pragma Priority(10);
end Controller;


task type Servers(Pri : System.Priority) is
  -- each instance of the task can have a
  -- different priority
  entry Service1(...);
  entry Service2(...);
  pragma Priority(Pri);
end Servers;
```

Priority Ceiling Locking

- Protected objects need to maintain the consistency of their data
- Mutual exclusion can be guaranteed by use of the priority model
- Each protected object is assigned a ceiling priority which is greater than or equal to the highest priority of any of its calling tasks
- When a task calls a protected operation, its priority is immediately raised to that of the protected object
- If a task wishing to enter a protected operation is running then the protected object cannot be already occupied

Ceiling Locking

- Each protected object is assigned a priority using a pragma
- If the pragma is missing, `Priority'Last` is assumed
- `Program_Error` is raised if the calling task's active priority is greater than the ceiling
- If an interrupt handler is attached to a protected operation and the wrong ceiling priority has been set, then the program becomes erroneous
- With ceiling locking, an effective implementation will use the thread of the calling task to execute not only the protected operation but also to execute the code of any other tasks that are released as a result of the call

Example of Ceiling Priority

```
protected Gate_Control is
   pragma Priority(28);
   entry Stop_And_Close;
   procedure Open;
private
   Gate : Boolean := False;
end Gate_Control;
```

```
protected body Gate_Control is
   entry Stop_And_Close
            when Gate is
   begin
      Gate := False;
   end;
   procedure Open is
   begin
      Gate := True;
   end;
end Gate_Control;
```

Example

- Assume task `T`, priority 20, calls `Stop_And_Close` and is blocked. Later task `S`, priority 27, calls `Open`. The thread executing `S` will undertake the following operations:
  - the code of `Open` for `S`
  - evaluate the barrier on the entry and note that `T` can now proceed
  - the code `Stop_And_Close` for `T`
  - evaluate the barrier again
  - continue with the execution of `S` after its call on the protected object
- There is no context switch

Active Priorities

- A task entering a protected operation has its priority raised

- A task's active priority might also change during:
  - task activation — a task inherits the active priority of the parent task which created it (to avoid priority inversion)
  - during a rendezvous — the task executing a rendezvous will inherit the active priority of the caller if it is greater than its current active priority
  - Note: no inheritance when waiting for task termination

Dispatching

- The order of dispatching is determined by the tasks' active priorities
- Default is preemptive priority based
- Not defined exactly what this means on a multi-processor system
- One policy defined by annex:
  `FIFO_Within_Priority`
- When a task becomes runnable it is placed at the back on the run queue for its priority; when it is preempted, it is placed at the front

Entry Queue Policies

- A programmer may choose the queuing policy for a task's entry queue and the select statement
- Two predefined policies: `FIFO_Queuing` (default) and `Priority_Queuing`
- With `Priority_Queuing` and the select statement, an alternative that is open and has the highest priority task queued (of all open alternatives) is chosen
- If there are two open with equal priority tasks, the one which appears textually first in the program is chosen
- Tasks are queued in active priority order, if active priority changes then no requeuing takes place; if the base priority changes, the task is removed and requeued

Dynamic Priorities

- Some applications require the base priority of a task to change dynamically: e.g., mode changes, or to implement dynamic scheduling schemes such as earliest deadline scheduling

Package Specification

```
with Ada.Task_Identification; use Ada;
package Ada.Dynamic_Priorities is
  procedure Set_Priority(Priority : System.Any_Priority;
    T : Task_Identification.Task_Id :=
    Task_Identification.Current_Task);
  function Get_Priority(T : T_Identification.Task_Id
     := Task_Identification.Current_Task)
     return System.Any_Priority;
     -- raise Tasking_Error if task has terminated
  -- Both raise Program_Error if a Null_Task_Id is passed
private
  -- not specified by the language
end Ada.Dynamic_Priorities;
```

Dynamic Priorities

- The effect of a change of base priorities should be as soon as practical but not during an abort deferred operation and no later than the next abort completion point

- Changing a task's base priority can affect its active priority and have an impact on dispatching and queuing

POSIX

- POSIX supports priority-based scheduling, and has options to support priority inheritance and ceiling protocols
- Priorities may be set dynamically
- Within the priority-based facilities, there are four policies:
  - FIFO: a process/thread runs until it completes or it is blocked
  - Round-Robin: a process/thread runs until it completes or it is blocked or its time quantum has expired
  - Sporadic Server: a process/thread runs as a sporadic server
  - OTHER: an implementation-defined
- For each policy, there is a minimum range of priorities that must be supported; 32 for FIFO and round-robin
- The scheduling policy can be set on a per process and a per thread basis

POSIX

- Threads may be created with a system contention option, in which case they compete with other system threads according to their policy and priority
- Alternatively, threads can be created with a process contention option where they must compete with other threads (created with a process contention) in the parent process
  - It is unspecified how such threads are scheduled relative to threads in other processes or to threads with global contention
- A specific implementation must decide which to support

Sporadic Server

- A sporadic server assigns a limited amount of CPU capacity to handle events, has a replenishment period, a budget, and two priorities
- The server runs at a high priority when it has some budget left and a low one when its budget is exhausted
- When a server runs at the high priority, the amount of execution time it consumes is subtracted from its budget
- The amount of budget consumed is replenished at the time the server was activated plus the replenishment period
- When its budget reaches zero, the server's priority is set to the low value

Other Facilities

POSIX allows:

- priority inheritance to be associated with mutexes (priority protected protocol= ICPP)
- message queues to be priority ordered
- functions for dynamically getting and setting a thread's priority
- threads to indicate whether their attributes should be inherited by any child thread they create

RT Java Threads and Scheduling

- There are two entities in Real-Time Java which can be scheduled:
  - `RealtimeThreads` (and `NoHeapRealtimeThread`)
  - `AsynEventHandler` (and `BoundAyncEventHandler`)

- Objects which are to be scheduled must
  - implement the `Schedulable` interface
  - specify their
    - `SchedulingParameters`
    - `ReleaseParameters`
    - `MemoryParameters`

Real-Time Java

- Real-Time Java implementations are required to support at least 28 real-time priority levels
- As with Ada and POSIX, the larger the integer value, the higher the priority
- Non real-time threads are given priority levels below the minimum real-time priority
- Note, scheduling parameters are bound to threads at thread creation time; if the parameter objects are changed, they have an immediate impact on the associated thread
- Like Ada and Real-Time POSIX, Real-Time Java supports a pre-emptive priority-based dispatching policy
- Unlike Ada and RT POSIX, RT Java does not require a preempted thread to be placed at the head of the run queue associated with its priority level

The Schedulable Interface

```
public interface Schedulable extends java.lang.Runnable
{
  public void addToFeasibility();
  public void removeFromFeasibility();

  public MemoryParameters getMemoryParameters();
  public void setMemoryParameters(MemoryParameters memory);


  public ReleaseParameters getReleaseParameters();
  public void setReleaseParameters(ReleaseParameters release);

  public SchedulingParameters getSchedulingParameters();
  public void setSchedulingParameters(
          SchedulingParameters scheduling);


  public Scheduler getScheduler();
  public void setScheduler(Scheduler scheduler);
}
```

Scheduling Parameters

```
public abstract class SchedulingParameters
{  public SchedulingParameters(); }

public class PriorityParameters extends SchedulingParameters
{
  public PriorityParameters(int priority);

  public int getPriority(); // at least 28 priority levels
  public void setPriority(int priority) throws
                          IllegalArgumentException;
  ...
}

public class ImportanceParameters extends PriorityParameters
{
  public ImportanceParameters(int priority, int importance);
  public int getImportance();
  public void setImportance(int importance);
  ...
}
```

RT Java: Scheduler

- Real-Time Java supports a high-level scheduler whose goals are:
  - to decide whether to admit new schedulable objects according to the resources available and a feasibility algorithm, and
  - to set the priority of the schedulable objects according to the priority assignment algorithm associated with the feasibility algorithm
- Hence, whilst Ada and Real-Time POSIX focus on static off-line schedulability analysis, Real-Time Java addresses more dynamic systems with the potential for on-line analysis

The Scheduler

```
public abstract class Scheduler
{
  public Scheduler();
  protected abstract void addToFeasibility(Schedulable s);
  protected abstract void removeFromFeasibility(Schedulable s);

  public abstract boolean isFeasible();
  // checks the current set of schedulable objects

  public boolean changeIfFeasible(Schedulable schedulable,
         ReleaseParameters release, MemoryParameters memory);

  public static Scheduler getDefaultScheduler();
  public static void setDefaultScheduler(Scheduler scheduler);

  public abstract java.lang.String getPolicyName();
}
```

The Scheduler

- The `Scheduler` is an abstract class
- The `isFeasible` method considers only the set of schedulable objects that have been added to its feasibility list (via the `addToFeasibility` and `removeFromFeasibility` methods)
- The method `changeIfFeasible` checks to see if its set of objects is still feasible if the given object has its release and memory parameters changed
- If it is, the parameters are changed
- Static methods allow the default scheduler to be queried or set
- RT Java does not require an implementation to provide an on-line feasibility algorithm

The Priority Scheduler

```
class PriorityScheduler extends Scheduler
{
  public PriorityScheduler()

  protected void addToFeasibility(Schedulable s);
  ...

  public void fireSchedulable(Schedulable schedulable);

  public int getMaxPriority();
  public int getMinPriority();
  public int getNormPriority();

  public static PriorityScheduler instance();
  ...
}
```

Standard preemptive priority-based scheduling

Other Facilities

- Priority inheritance and ICCP (called priority ceiling emulation)
- Support for aperiodic threads in the form of processing groups; a group of aperiodic threads can be linked together and assigned characteristics which aid the feasibility analysis

Summary

- A scheduling scheme defines an algorithm for resource sharing and a means of predicting the worst-case behaviour of an application when that form of resource sharing is used.
- With a cyclic executive, the application code must be packed into a fixed number of minor cycles such that the cyclic execution of the sequence of minor cycles (the major cycle) will enable all system deadlines to be met
- The cyclic executive approach has major drawbacks many of which are solved by priority-based systems
- Simple utilization-based schedulability tests are not exact

Summary

- Response time analysis is flexible and caters for:
  - Periodic and sporadic processes
  - Blocking caused by IPC
  - Cooperative scheduling
  - Arbitrary deadlines
  - Release jitter
  - Fault tolerance
  - Offsets
- Ada, RT POSIX and RT Java support preemptive priority-based scheduling
- Ada and RT POSIX focus on static off-line schedulability analysis, RT Java addresses more dynamic systems with the potential for on-line analysis