Implementação de escalonamento por loteria no *xv6* com análise de desempenho

João Pedro Winckler Bernardi[1]

[1]Universidade Federal da Fronteira Sull (UFFS)
89802-260 – Chapecó – SC – Brasil

jpwbernardi@hotmail.com

Abstract. *In a lottery scheduler, for each process is given a certain amount of tickets. A random ticket is selected and the process that owns that ticket is executed. This report shows the lottery sceduler's BIT – Binary Indexed Tree – implementation on xv6 and analyzes if works as required.*

Resumo. *No escalonamento por loteria, cada processo recebe uma quantidade de bilhetes. O processo com o bilhete sorteado é executado. Nesse relatório, apresentamos uma implementação feita com uma BIT – Binary Indexed Tree – para o escalonador por loteria no sistema operacional xv6 e analisamos seu funcionamento.*

1. Introdução

Os sistemas operacionais executam vários processos ao mesmo tempo, ou, pelo menos, é essa a impressão que temos. Um CPU não consegue executar mais de um processo simultaneamente, o que ocorre é que todo processo está competindo para ganhar fatias de tempo do processador. O escalonador é o responsável pela escolha do processo a ser executado. Na troca de processos, o estado do processo atual deve ser salvo para que na próxima vez que ele for escolhido esses dados sejam recarregados e ele possa continuar a execução exatamente do momento que parou.

O método que o escalonador vai usar para escolher o próximo processo é chamado de algoritmo de escalonamento. Esse relatório apresenta o funcionamento do escalonamento por loteria com complexidade de $O(\log N)$, sendo N a quantidade máxima de processos, e sua implementação no sistema operacional xv6.

2. Escalonamento por loteria

Esse algoritmo de escalonamento funciona da seguinte forma: cada processo recebe uma quantidade de bilhetes, então sorteia-se um bilhete e o processo dono daquele bilhete é executado. Embora todos os processos tenham bilhetes, só os processos prontos(*RUNNABLE* no xv6) devem ser sorteados. Assim surgiu a ideia do acumulo de bilhetes.

Associamos cada processo a uma posição i de um vetor que guarda a soma de bilhetes do processo na posição 0 até a posição i. Por exemplo, os processos A, B, C, D, E com 10, 20, 5, 1 e 13 bilhetes, respectivamente, estão prontos. Por tanto, temos 49 bilhetes que podem ser sorteados e nosso vetor estaria da seguinte forma:

Tabela 1. Exemplo do método de bilhetes acumulados 1

A	B	C	D	E
10	30	35	36	49

Isso significa que os bilhetes de A estão entre 1 e 10, os de B entre 11 e 30, os de C entre 31 e 35, o de D é 36 e os de E entre 37 e 49. Então, suponhamos que o bilhete sorteado foi o 15, ou seja, o processo B será executado. Mas durante sua execução foi bloqueado, isso implica que B não pode mais ser executado enquanto não estiver *RUNNABLE*. Portanto, no vetor onde guardamos os acúmulos, retiramos os bilhetes de B. O vetor de acúmulos estaria da seguinte maneira:

Tabela 2. Exemplo do método de bilhetes acumulados 2

A	B	C	D	E
10	10	15	16	29

Agora, os bilhetes de A estão entre 1 e 10, B não tem bilhete, os bilhetes de C estão entre 11 e 15, o de D é 16 e os de E entre 17 e 29. Note que é apenas no vetor de acúmulos que os bilhetes são retirados, a informação que o B tem 20 bilhetes continua guardada, pois quando ele estiver pronto novamente, devolveremos os bilhetes para o vetor de acumulos.

Trabalhando com o acumulo de bilhetes, fica simples de se implementar o sorteio de um bilhete, pois é um intervalo contínuo. Se tivéssemos atribuído números fixos aos bilhetes, quando um processo fosse bloqueado geraria uma buraco no intervalo de bilhetes a serem sorteados e teríamos que tratar para o sorteio não considerar esses buracos.

Assim, chega-se a outro problema: fazer o acúmulo de bilhetes de forma eficiente. Pois, como apresentado até agora, a ideia simples para fazer o cálculo desses acumulos seria percorrendo o vetor e atualizando cada posição, uma complexidade de $O(n)$, sendo n a quantidade máxima de processos permitidos, para cada vez que fosse necessário atualizar a quantidade de bilhetes um processo. Porém, existe uma estrutura de dados chamada BIT – Binary Indexed Tree – que faz essa operação com complexidade $O(\log n)$.

Por fim, o último método utilizado para manter a complexidade baixa é buscar o processo com o bilhete sorteado através de uma busca binária.

```
int bsearch(int ticket){
  int l = 1, h = NPROC, m;
  while (l < h) {
      m = l + (h - l) / 2;
      if (ticount(m) >= ticket) h = m;
      else l = m + 1;
  }
  return l - 1;
}
```

3. Binary Indexed Tree(BIT)

Inventada por Peter M. Fenwick em 1994, a BIT é uma estrutura de dados simples para implementar tabelas de frequências cumulativas. A implementação foi feita através de um vetor, onde cada posição guarda um acumulo parcial. Na implementação, a BIT é representada pelo vetor $stickets$. O vetor $idstack$ é uma pilha estática que guarda as posições da BIT não associadas a processos, tp é em que posição se encontra o topo da pilha. É importante dizer que a posição 0 da BIT não é utilizada.

```
struct {
  int stickets[NPROC + 1];
  int idstack[NPROC], tp;
  struct spinlock lock;
  struct proc proc[NPROC];
} ptable;
```

De uma forma mais genérica, o elemento da posição i da BIT $stickets$ é responsável pelos elementos no intervalo [$i - (i$ AND $-i) + 1$, i] e $stickets[i]$ guarda o acumulo dos bilhetes $\{i - (i$ AND $-i) + 1, i - (i$ AND $-i) + 2, i - (i$ AND $-i) + 3$, ... , i$\}$, como mostrado na Figura 1.

Figura 1. Relação das posições dependentes

Para obtermos o acúmulo de bilhetes de uma posição, usamos a função *ticount*. Para atualizarmos uma posição da BIT com algum valor, usamos a função *uptick*.

```
int ticount(int i){
  int count = 0;
  for (; i > 0; i -= i & -i)
    count += ptable.stickets[i];
  return count;
}

void uptick(int i, int value){
  for (; i <= NPROC; i += i & -i)
```

```
    ptable.stickets[i] += value;
}
```

4. Mudanças no *xv6*

Inicialmente, o pid de um processo era definido por uma variável global. Toda vez que um processo recebia um pid, a global era incrementada. Porém, para facilitar a implementação e otimização de código, o pid representa o índice da BIT associado ao processo. Toda vez que um processo termina, ele devolve seu pid para a pilha $idstack$. Como cada processo tem um pid único que varia de 1 a n, então também podemos usar o pid para a indexação do processo no vetor de processos $ptable.proc$, obviamente subtraindo 1, pois as posições de $ptable.proc$ são de 0 a $n - 1$. Antes, quando um processo era criado, para achar uma posição livre no vetor de processos, ele era percorrido linearmente. Agora, só retiramos da pilha um pid disponível, o que é realizado em tempo $O(1)$.

```
static struct proc* allocproc(void){
 ...
  acquire(&ptable.lock);
  if (ptable.tp > 0) {
    i = ptable.idstack[--ptable.tp];
    p = &ptable.proc[i - 1];
    goto found;
  }
  release(&ptable.lock);
  return 0;
found:
  p->pid = i;
 ...
  return p;
}
```

Toda vez que o *xv6* é inicializado, a função $clean$ é chamada. Ela é responsável por inicializar a pilha, ou seja, colocar todas as posições disponíveis da BIT na pilha, e zerar a BIT.

```
void clean() {
  acquire(&ptable.lock);
  for (ptable.tp = 0; ptable.tp < NPROC; ptable.tp++)
    ptable.idstack[ptable.tp] = NPROC - ptable.tp;
  memset(ptable.stickets, 0, sizeof (ptable.stickets));
  release(&ptable.lock);
}
```

E a função *scheduler* é a responsável pela mudança de processo em execução. Primeiro verificamos a quantidade de bilhetes disponíveis para serem soreados. Se essa quantidade for diferente de 0, ou seja, houver algum bilhete para ser sorteado, sorteamos um bilhete entre 1 e a quantidade de bilhetes. A busca binária retorna a posição no vetor *ptable.proc* do processo que tinha o bilhete sorteado. Então, retiramos os bilhetes desse processo e mudamos seu estado para *RUNNING*.

```
void scheduler(void){
  ...
  for(;;){
    sti();
    acquire(&ptable.lock);
    if ((qttytickets = ticount(NPROC)) != 0) {
      p = &ptable.proc[bsearch(rand() % qttytickets + 1)];
      if (p->state == RUNNABLE) {
        uptick(p->pid, -p->tickets);
        proc = p;
        switchuvm(p);
        p->state = RUNNING;
        swtch(&cpu->scheduler, proc->context);
        switchkvm();
        proc = 0;
      }}
    release(&ptable.lock);
  }}
```

Os bilhetes do processo são devolvidos quando o tempo dele no CPU termina e ele volta para o estado *RUNNABLE*.

```
void yield(void){
  acquire(&ptable.lock);
  proc->state = RUNNABLE;
  uptick(proc->pid, proc->tickets);
  sched();
  release(&ptable.lock);
}
```

5. Análise de desempenho

Toda vez que o escalonador busca um processo para ser executado, é chamado a busca binária, que tem complexidade $O(\log n)$, porém, para acessar o valor do acúmulo numa posição i, a complexidade é também $O(\log n)$. Ou seja, a complexidade total para escolher um processo seria $O(\log(n \cdot \log n))$. Desenvolvendo a conta:

$$O(\log(n \cdot \log n)) = O(\log n \cdot \log(\log n)) = O(\log n)$$

6. Análise de testes

Para testar o funcionamento do que foi implementado, criamos um arquivo $schedtest$ no $xv6$. Quando executado, cria o máximo de processos possíveis. Cada processo decrementa uma variável que começa em aproximadamente 10^8 e, quando essa variável chega a 0, o processo acaba. Para obter os resultados do teste, foi modificado a função $exit()$ para que quando ela fosse chamada, exibisse a quantidade de tickets do processo que acabou. Cada processo tem uma quantidade de bilhetes diferentes. Esse teste foi realizado 10 vezes com processos de mesma quantidade de bilhete.

O resultado obtido foi está apresentado nos seguintes gráficos, separados para melhor visualização. O eixo x é o número do teste e o eixo y é a ordem em que ele

acabou. A quantidade de bilhetes de um processo está entre parenteses na legenda. Por exemplo, o processo 1 em todos os testes foi o 61º processo a terminar.

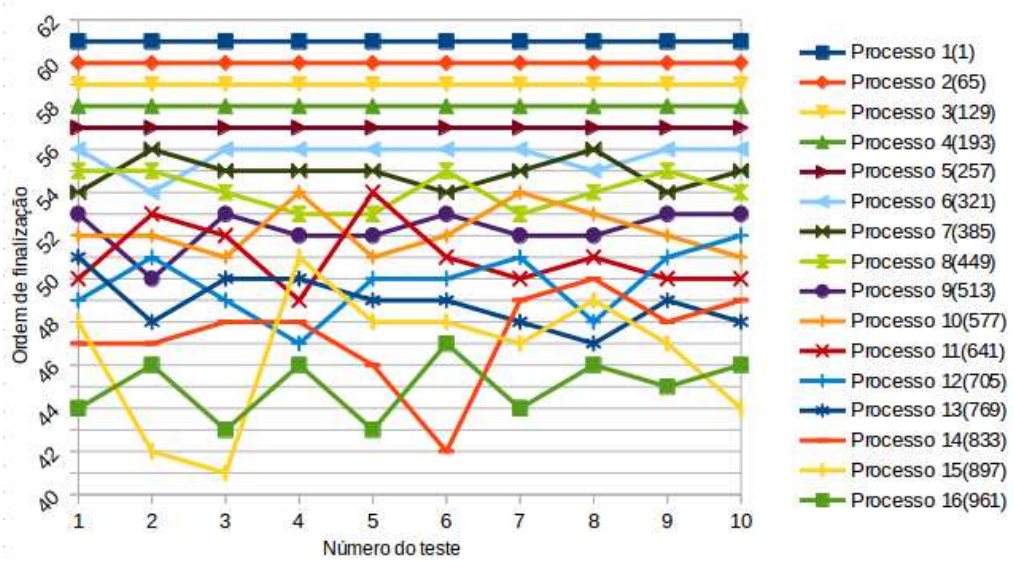

Figura 2. Relação da posições dependentes

7. First Page

The first page must display the paper title, the name and address of the authors, the abstract in English and "resumo" in Portuguese ("resumos" are required only for papers written in Portuguese). The title must be centered over the whole page, in 16 point boldface font and with 12 points of space before itself. Author names must be centered in 12 point font, bold, all of them disposed in the same line, separated by commas and with 12 points of space after the title. Addresses must be centered in 12 point font, also with 12 points of

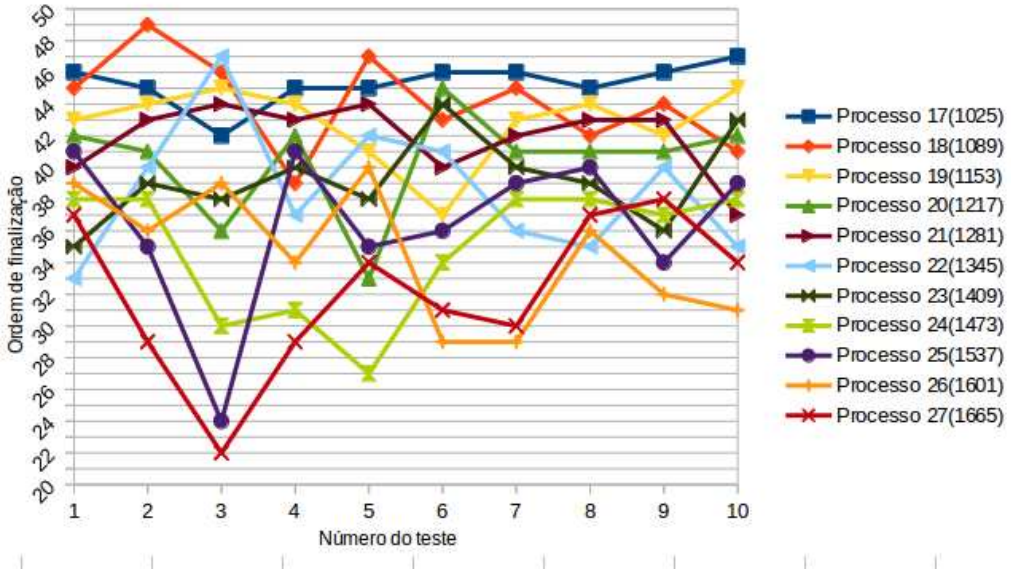

Figura 3. Relação das posições dependentes

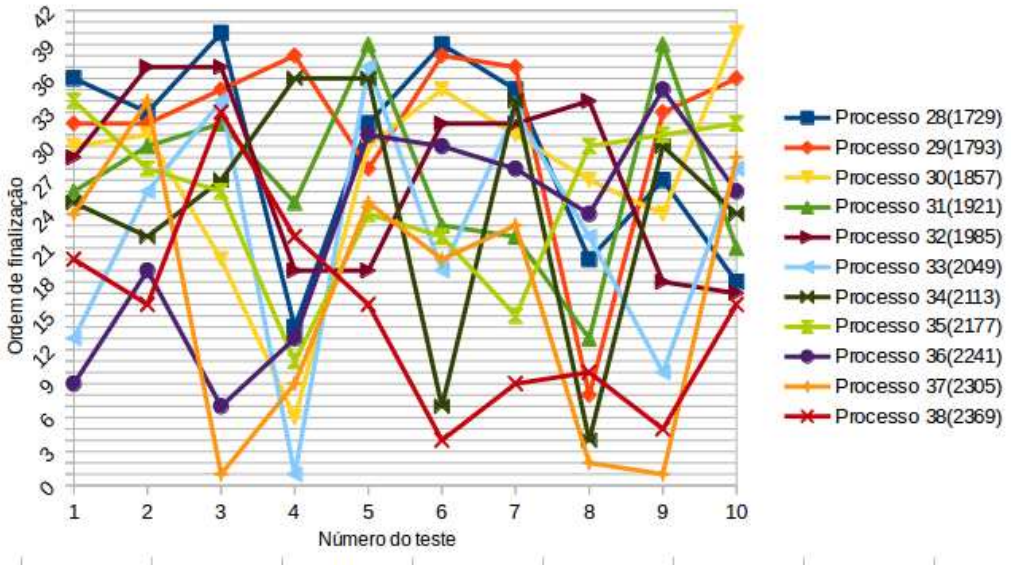

Figura 4. Relação das posições dependentes

space after the authors' names. E-mail addresses should be written using font Courier New, 10 point nominal size, with 6 points of space before and 6 points of space after.

The abstract and "resumo" (if is the case) must be in 12 point Times font, indented 0.8cm on both sides. The word **Abstract** and **Resumo**, should be written in boldface and must precede the text.

8. CD-ROMs and Printed Proceedings

In some conferences, the papers are published on CD-ROM while only the abstract is published in the printed Proceedings. In this case, authors are invited to prepare two final

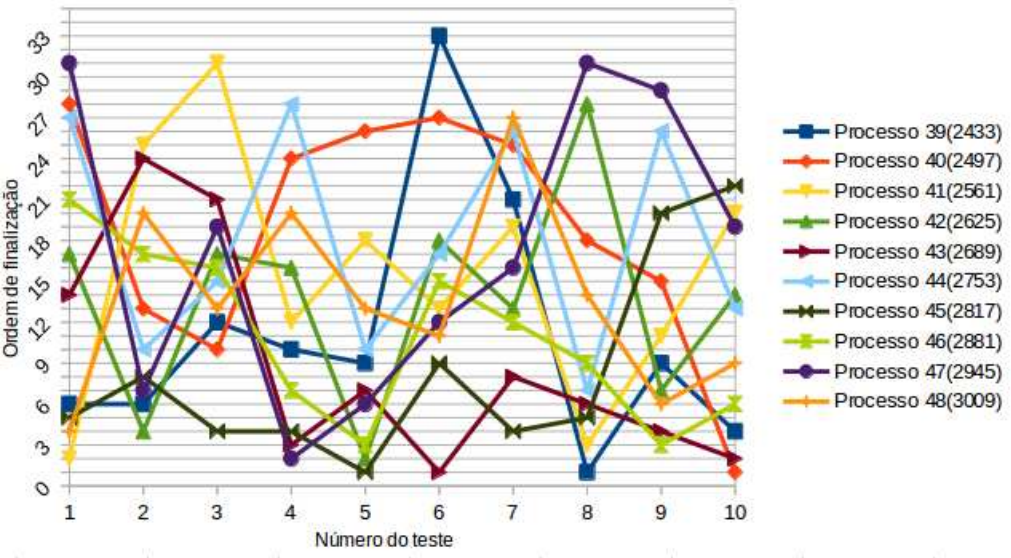

Figura 5. Relação das posições dependentes

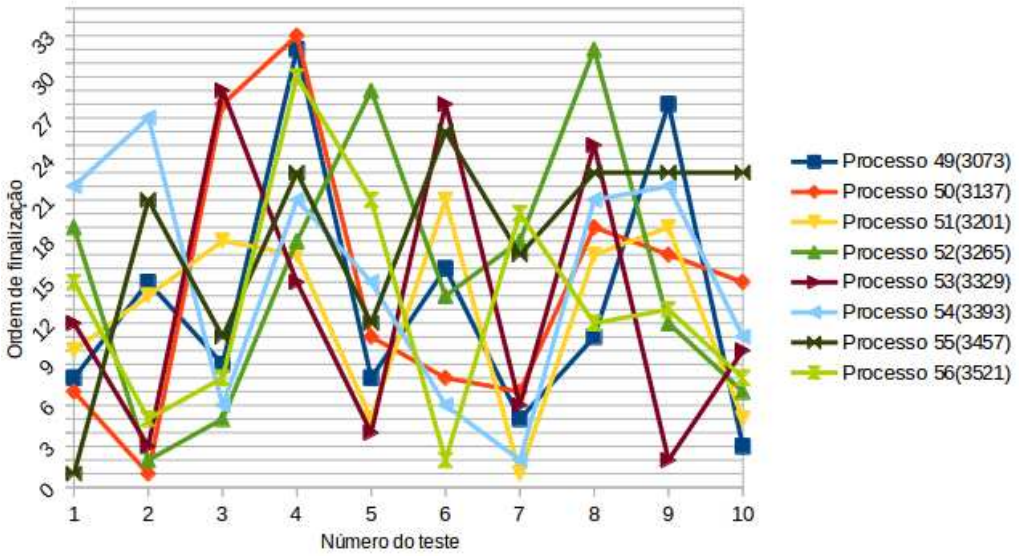

Figura 6. Relação das posições dependentes

versions of the paper. One, complete, to be published on the CD and the other, containing only the first page, with abstract and "resumo" (for papers in Portuguese).

9. Sections and Paragraphs

Section titles must be in boldface, 13pt, flush left. There should be an extra 12 pt of space before each title. Section numbering is optional. The first paragraph of each section should not be indented, while the first lines of subsequent paragraphs should be indented by 1.27 cm.

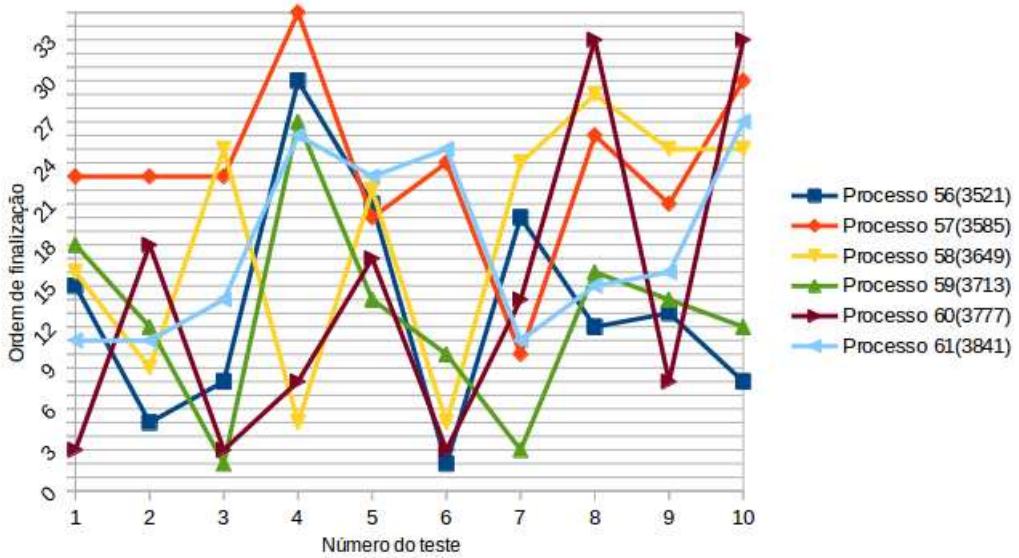

Figura 7. Relação das posições dependentes

9.1. Subsections

The subsection titles must be in boldface, 12pt, flush left.

10. Figures and Captions

Figure and table captions should be centered if less than one line (Figure 8), otherwise justified and indented by 0.8cm on both margins, as shown in Figure 9. The caption font must be Helvetica, 10 point, boldface, with 6 points of space before and after each caption.

Figura 8. A typical figure

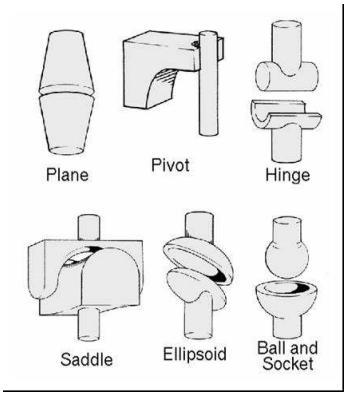

Figura 9. This figure is an example of a figure caption taking more than one line and justified considering margins mentioned in Section 10.

In tables, try to avoid the use of colored or shaded backgrounds, and avoid thick, doubled, or unnecessary framing lines. When reporting empirical data, do not use more decimal digits than warranted by their precision and reproducibility. Table caption must be placed before the table (see Table 1) and the font used must also be Helvetica, 10 point, boldface, with 6 points of space before and after each caption.

Tabela 3. Variables to be considered on the evaluation of interaction techniques

	Chessboard top view	Chessboard perspective view
Selection with side movements	6.02 ± 5.22	7.01±6.84
Selection with in-depth movements	6.29±4.99	12.22±11.33
Manipulation with side movements	4.66± 4.94	3.47±2.20
Manipulation with in-depth movements	5.71 ±4.55	5.37 ±3.28

11. Images

All images and illustrations should be in black-and-white, or gray tones, excepting for the papers that will be electronically available (on CD-ROMs, internet, etc.). The image resolution on paper should be about 600 dpi for black-and-white images, and 150-300 dpi for grayscale images. Do not include images with excessive resolution, as they may take hours to print, without any visible difference in the result.

12. References

Bibliographic references must be unambiguous and uniform. We recommend giving the author names references in brackets, e.g. [Knuth 1984], [Boulic and Renault 1991], and [Smith and Jones 1999].

The references must be listed using 12 point font size, with 6 points of space before each reference. The first line of each reference should not be indented, while the subsequent should be indented by 0.5 cm.

Referências

Boulic, R. and Renault, O. (1991). 3d hierarchies for animation. In Magnenat-Thalmann, N. and Thalmann, D., editors, *New Trends in Animation and Visualization*. John Wiley & Sons ltd.

Knuth, D. E. (1984). *The TEX Book*. Addison-Wesley, 15th edition.

Smith, A. and Jones, B. (1999). On the complexity of computing. In Smith-Jones, A. B., editor, *Advances in Computer Science*, pages 555–566. Publishing Press.