Capítulo 5

Trabalhando com o Processador

Quando temos vários processos trabalhando de forma concorrente, ou seja, um competindo com o outro para executar suas tarefas, devemos ter um critério de seleção e solução para um correto compartilhamento do processador. Existem diversas técnicas e algoritmos implementados para o uso do mesmo. Nesse capítulo veremos os principais tipos de escalonadores de processos, que são os mecanismos de seleção de como e qual processo será executado.

5.1 Não-preemptivo

Podemos definir como escalonamento não-preemptivo aquele utilizado nos primeiros sistemas operacionais multiprogramáveis, que faziam uso de processamento batch, onde um processo ganhava o direito de usar a UCP e continuava com esse direito, sem que outro processo pudesse lhe tirar esse recurso. Nesse tipo de escalonamento temos os algoritmos específicos, e vamos estudar cada um com um algoritmo em linguagem C, por ser mais fácil a compreensão.

5.1.1 SJF (Trabalho mais curto primeiro)

Nesse tipo de escalonamento, como o próprio nome sugere (SJF – Shortest Job First), o trabalho que tenha menor tempo de execução tem prioridade sobre os outros, assim ele ocupa a UCP por um curto período de tempo e depois libera seu uso para outros processos. A grande dificuldade é determinar com precisão quanto tempo um processo ocupa da UCP, para poder ordenar a fila de processos. Para tornar a idéia mais clara vamos a um código simplista escrito em linguagem C.

```
#include <stdio.h>

int c=0, tempo[100];

int sjf(int a);

void main()
```

```
{
  int i=1, proc, b;

  while (i == 1)
  {
    printf(淣igite o tempo de execucao do processo %i, ou 0 para fim:    c);
    scanf(   i   &tempo[c]);

    if (tempo[c] == 0)
    {
      i = 0;
      break;
    }

    if (c > 0)
      sjf(c);

    c++;

  }

  if (c > 0)
  {
    for(b=0; b<c; b++)
      printf(   \nOrdem final dos processos: %i   tempo[b]);
  }
  else
    printf(哲ao existem processos para execucao.   ;
}

int sjf(int a)
{
  int l, swap;

  for(l=c; l>1; l--)
```

```
  {


    if (tempo[a] < tempo[l])
    {
      swap = tempo[l];
      tempo[l] = tempo[a];
      tempo[a] = swap;
      a=l;
    }
  }


  return;
}
```

Podemos notar com clareza que o algoritmo depende totalmente de saber com precisão o tempo de execução de cada processo, o que é muito difícil prever como fizemos com esse "simulador", já que mesmo sabendo quais as instruções de um processo e quantos clocks cada instrução consome, dificilmente seria possível calcular esse tempo com precisão.

Quando um processo é chamado para ser executado, a rotina "sjf" é acionada, então o processo é colocado no seu devido lugar na fila, quer dizer, se ele for menor que o processo anterior ele "pula" na frente e assim subseqüentemente até chegar a uma posição que o processo anterior a ele seja realmente menor.

entra f050101.jpg

Figura 5.1.1 – O processo com menor tempo acessa a UCP primeiro.

5.1.2 FIFO (Primeiro a Entrar, Primeiro a Sair)

Em algoritmos FIFO (First In First Out), quando um processo é acionado para ser executado, ele entra na fila de processos e ocupa todo o processador e seus recursos até que seja executado totalmente.

Veremos mais adiante em nosso próprio sistema operacional a implementação prática do FIFO, que é um tanto quanto simples e fácil de ser codificado por um programador. De momento veremos um exemplo em linguagem C.

```
#include <stdio.h>

int c=0, tempo[100];

void main()

{

  int i=1, proc, b;
```

```
while (i == 1)
{
  printf(淚igite um numero para identificar esse processo %i, ou 0 p/ fim:      c);
  scanf(   i   &tempo[c]);

  if (tempo[c] == 0)
  {
    i = 0;
    break;
  }

  c++;

}

if (c > 0)
{
  for(b=0; b<c; b++)
    printf(    \nOrdem final dos processos: %i    tempo[b]);
}
else
  printf(晢ao existem processos para execucao.   ;

}
```

Nada pode ser mais simples que isso em termos de escalonadores, porque o único trabalho do escalonador é receber um processo e encaminhá-lo ao final da final.

entra f050102.jpg

Figura 5.1.2 – O Processo 1 é o primeiro a entrar e o primeiro a sair.

5.1.3 Escalonamento Cooperativo

Nesse escalonamento cabe a cada processo em execução liberar o uso do processador, tornando o uso do processador um processo de distribuição utilizado em sistemas multiprogramáveis.

Como o sistema operacional não influi nessa chamada “passagem de bastão” de um processo para o outro, podem ocorrer problemas graves de monopólio excessivo de um processo ou até mesmo um “looping” infinito do mesmo.

```
#include <stdio.h>

int c=0, tempo[100];

void main()

{

  int i=1, proc, b=1, a;


  while (i == 1)

  {

    printf(")Digite um numero para identificar esse processo %i, ou 0 p/ fim:    c);

    scanf(   i   &tempo[c]);


    if (tempo[c] == 0)

    {

      i = 0;

      break;

    }


    c++;


  }


 if (c > 0)

 {

       for(a=0; a<c; a++)

       {

         while(b!=0)

              {

        printf(   \nProcesso em execucao: %i, digite 0 p/ liberar UCP: %i   tempo[a]);

                 scanf(   i   &b);

              }

              b=1;

       }
```

```
  }
  else
    printf(哲ao existem pr  ocessos para execucao.  ;


}
```

Quando, no nosso simulador, digitamos o 0, imaginamos que em um programa de verdade ele está liberando o uso do processador, o que precisa ser feito com atenção, para que um processo não faça um possível uso "imprevisível" do processador, mas que ele mesmo se policie quanto ao tempo que ele fará uso.

5.2 Preemptivo

Quando processos concorrem para a execução e o processador interrompe um outro para que determinado outro processo entre em execução, temos uma situação que pode ser definida como escalonamento do tipo preemptivo.

Determinado critério é adotado (tempo, prioridade etc.), estabelecendo a implementação de diversos tipos de algoritmos de escalonadores não preemptivos, onde características dos processos como por exemplo o tempo, um campo marcando prioridade, ou outra característica estabelece quando um processo deverá "parar" para que outro entre em execução.

O termo overhead é muito usado em escalonamento preemptivo, e ocorre quando um processo alterna para outro na UCP, gerando uma lentidão no sistema.

5.2.1 Circular ou Robin Round

Esse tipo de escalonador trabalha com os processos entrando numa fila circular, sempre entrando um processo no final da fila. Acontece que para um processo parar sua execução, ele precisa de um tempo determinado (também chamado de quantum), em geral algo em torno de 100 e 300 ms [4], assim o próximo processo do círculo entra em execução pelo mesmo tempo e segue-se a fila.

Esse tipo de escalonador trabalha, como podemos observar, com fatias de tempo que nada mais são do que os milessegundos que cada processo ocupa. Quando o último processo da fila é executado, volta-se para o começo da fila, daí então o termo fila circular. Vamos ao código em C.

```
#include <stdio.h>

int c=0, tempo[100];

void main()

{

  int i=1, b, a, proc, ts, ciclos;
```

```
printf(鞴uantos ms por fatia de tempo (time  -slice):    ;
scanf(   i   &ts);
printf(沢igite a quantidade de ciclos:    ;
scanf(   i   &ciclos);


while (i == 1)
{
  printf(沢igite um numero para identifica  r esse processo %i, ou 0 p/ fim:    c);
  scanf(   i   &tempo[c]);


  if (tempo[c] == 0)
  {
    i = 0;
    break;
  }


  c++;


}


if (c > 0)
{
  for(a=0; a<ciclos; a++)
  {
    for(b=0; b<c; b++)
    {
      printf(   i   tempo[b]);
      for(i=0; i<ts; i++)
        printf(   -   ;
    }
    printf(   \n   ;
  }
}
else
```

```
    printf("Nao existem processos para execucao.  ;

}
```

Nos laços de nosso programa, vemos o símbolo "-" representando os milessegundos, ou unidades de tempo quaisquer que sejam adotadas. Quando um processo chega ao limite de unidades de tempo, ele passa sua vez de uso da UCP para o próximo, e espera até que ele retome a condição de processo que faz uso da UCP.

É claro que temos aqui um código bem simplificado, mas que para fins didáticos é mais que completo para o entendimento de algo que costuma ser muito mistificado, tornando assim sua compreensão muito mais fácil.

entra f050201.jpg

Figura 5.2.1 – Enquanto não terminam, eles retornam ao fim da fila.

5.2.2 Escalonamento Por Prioridades

No escalonamento por prioridades solucionamos um problema clássico que o escalonamento circular não resolveria. Quando temos um processo que trabalha a maior parte do tempo com E/S (Entrada e Saída) concorrendo com um outro processo que faz uso em sua maior parte de código da UCP, o que acontece é que o processo E/S fica muito tempo no estado de espera e deveria ter alguma prioridade com relação ao processo que faz maior uso da UCP, é então que entra o escalonamento por prioridades.

Quando um processo entra na fila, seu "flag" de prioridade é avaliado, e, se ele for maior que o anterior, passa para frente na fila de execução. Assim, ele passará a ocupar unidades proporcionais de tempo para que o uso da UCP seja dividido de acordo com a necessidade de cada um, compartilhando com maior igualdade os recursos. Vale lembrar que o conceito de prioridade pode ser estático ou dinâmico, sendo que em um a prioridade do processo não se altera desde quando ele entra na fila e no outro a prioridade do processo pode se alterar conforme a necessidade.

Para exemplificar com maior clareza esses conceitos, foi adotado um código escrito por Carlos Henrique de Oliveira, com o qual ele conseguiu demonstrar detalhadamente o escalonamento por prioridades.

```
/*  Programado por Carlos Henrique Pereira de Oliveira  */

/*  http://planeta.terra.com.br/lazer/universokk/      */

/*  kkrj@terra.com.br                                  */


#include <stdio.h>

#include <conio.h>


/* Define o numero maximo de processos concorrentes */
```

```
#define MAXPRO 3

/* Define o numero maximo de entrada-e-saidas de um processo */
#define MAXIO 6

/* Estrutura um contexto de software hipotetico do processo */
typedef struct
{
  int PID;
      int PRIO;
      int CPUTIME;
      int REALTIME;
      int ELAPSED;
      int IO[MAXIO];
      int REALIO;
      int STATE;
} Tprocesso;

/* Estrutura uma lista */
typedef struct
{
  Tprocesso NO[MAXPRO];
  int QTD;
}Tlista;

/* Funcao que inicia uma lista */
void INICIALISTA (Tlista *L)
{
  L->QTD=0;
}

/* Funcao que verifica se uma lista esta cheia */
int LISTACHEIA (Tlista *L)
{
  if (L->QTD == MAXPRO)
```

```
    return 1;


  return 0;
}

/* Funcao que verifica se uma lista esta vazia */
int LISTAVAZIA (Tlista *L)
{
  if (L->QTD == 0)
    return 1;


  return 0;
}

/* Funcao que inicia banco de PID unico */
void INICIAPID (int *L)
{
  int CONTA, NUM;
  CONTA = 0;
  NUM = 100;
  while (CONTA < 12)
  {
    L[CONTA] = 1;
    CONTA++;
    L[CONTA] = NUM;
    CONTA++;
    NUM++;
  }
}

/* Funcao cria um novo processo verificando erro de entrada de dados */
void CRIAPROCESSO (Tprocesso *L)
{
  int CORRETO, CONTA, N;
```

```
CORRETO = 0;
while (CORRETO == 0)
{
  printf (   \n\nPID: %d   ,L ->PID);
  printf (   \n\n    Entre com o CPU-TIME (em ut):    ;
  scanf (   d   ,&L    ->CPUTIME);
  clrscr();
  if (L->CPUTIME <= 0)
  {
    printf (   \n\n %d - CPU-TIME deve ser > que 0 ut    ,L ->CPUTIME);
  }
  else
  {
    L->REALTIME = 0;
    L->ELAPSED = 0;
    CORRETO = 1;
  }
}

CORRETO = 0;
while (CORRETO == 0)
{
  printf (   \n\nPID: %d   ,L ->PID);
  printf (   \nCPU-TIME: %d   ,L ->CPUTIME);
  printf (   \n\n    Niveis de PRIORIDADES : 1 2 3 4 5   ;
  printf (   \n\n    Entre com a PRIORIDADE:    ;
  scanf (   d   ,&L    ->PRIO);
  clrscr();
  if ((L->PRIO < 1) || (L->PRIO > 5))
  {
    printf (   \n\n    +++ %d - PRIORIDADE invalida +++   ,L ->PRIO);
  }
  else
  {
    CORRETO = 1;
```

```
  }
}

CONTA = 0;
CORRETO = 0;
L->REALIO = 0;
L->IO[CONTA] = 1;
L->IO[MAXIO - 1] = 0;
while ((CORRETO == 0) && (L->IO[CONTA] != 0))
{
  printf (   \n\nPID: %d   ,L ->PID);
  printf (   \nCPU-TIME: %d   ,L ->CPUTIME);
  printf (   \nPRIORIDADE: %d   ,L ->PRIO);
  printf (   \nTempos de E/S:   ;

  for (N = 0 ; N < CONTA ; N++)
  {
    printf (   d   ,L   ->IO[N]);
  }

  printf (   \n\n   Entre com os tempos quais as E/S serao processadas   ;
  printf (   \n   ou 0 para encerrar a definicao de E/S:   ;
  scanf (   d   ,&L   ->IO[CONTA]);
  clrscr();

  if (L->IO[CONTA] >= 0)
  {
    if (L->IO[CONTA] == 0)
      {
CORRETO = 1;
      }
      CONTA++;
  }
  else
  {
```

```
        printf (    \n\n %d - Tempo E/S deve ser > que 0 ut     ,L ->IO[CONTA]);
      }
    }
}

/* Funcao que ordena tempos de E/S, exclui tempos duplicados e carimba o estado do processo como PRONTO */
/* Considere que: (+1) Pronto e (-1) Espera */
void ORDENAES (Tprocesso *L)
{
  int CORRETO, CONTA, AJUDA, N;

  CORRETO = 0;
  L->STATE = +1;
  while (CORRETO == 0)
  {
    CORRETO = 1;
    for (CONTA = 0 ; L->IO[CONTA] != 0 ; CONTA++)
    {
      AJUDA = CONTA + 1;
      while (L->IO[AJUDA] != 0 )
      {
        if (L->IO[CONTA] > L->IO[AJUDA])
        {
        CORRETO = 0;
        N = L->IO[CONTA];
        L->IO[CONTA] = L->IO[AJUDA];
        L->IO[AJUDA] = N;
        }

      if (L->IO[CONTA] == L->IO[AJUDA])
        {
        CORRETO = 0;
        N = AJUDA;
        while (L->IO[N] != 0)
          {
```

```
            L->IO[N] = L->IO[N+1];
            N++;
            }
          }

        AJUDA++;
        }
      }
    }
  clrscr();
  printf (   \n\nPID: %d   ,L ->PID);
  printf (   \nCPU-TIME: %d   ,L ->CPUTIME);
  printf (   \nPRIORIDADE: %d   ,L ->PRIO);
  printf (   \nTempos de E/S:    ;
  for (N = 0 ; L->IO[N] != 0 ; N++)
  {
    printf (   d    ,L    ->IO[N]);
  }
}

/* Funcao que insere um processo numa lista determinada */
int INSERENO (Tlista *L, Tprocesso NOVO)
{
  if (LISTACHEIA (L))
    return 0;

  L->NO[L->QTD]=NOVO;
  L->QTD++;
  return 1;
}

/* Funcao que retira um processo de uma lista determinada */
void RETIRANO (Tlista *L)
{
  L->QTD--;
```

```
}

/* Funcao que orgazina prioridade dentro da lista */
void ORGANIZAPRIO (Tlista *L)
{
  int CONTA, AJUDA;
  Tprocesso TEMP;

  if (!LISTAVAZIA (L))
  {
    for (CONTA = 0 ; CONTA < L->QTD ; CONTA++)
    {
      for (AJUDA = CONTA+1 ; AJUDA < L->QTD ; AJUDA++)
      {
        if (L->NO[CONTA].PRIO > L->NO[AJUDA].PRIO)
        {
        TEMP = L->NO[CONTA];
        L->NO[CONTA] = L->NO[AJUDA];
        L->NO[AJUDA] = TEMP;
        }
      }
    }
  }

}

/* ***************************************** */
/*      Processa cada ciclo de clock         */
/* ***************************************** */
void PROCESSADOR (Tlista *P, Tlista *E, int *CLOCK)
{
  int CONTA, AJUDA, CORRETO;
  Tprocesso TEMP;

  /* Se o processo terminou */
```

```
if (P->NO[P->QTD - 1].REALTIME == P->NO[P->QTD - 1].CPUTIME)
{
  RETIRANO (P);
}

/* Se realtime do processo MENOR que cputime */
if (P->NO[P->QTD - 1].REALTIME < P->NO[P->QTD - 1].CPUTIME)
{
  CONTA = 0;
  CORRETO = 1;
  /* Enquanto IO for diferente de zero */
  while (P->NO[P->QTD - 1].IO[CONTA] != 0)
  {
    /* Se o tempo de IO for o mesmo do clock */
    if (P->NO[P->QTD - 1].IO[CONTA] == *CLOCK)
    {
      CORRETO = 0;
    P->NO[P->QTD - 1].REALIO = 5;
    TEMP = P->NO[P->QTD - 1];
    RETIRANO (P);
    INSERENO (E, TEMP);

    P->NO[P->QTD-1].REALTIME++;

    for (AJUDA = P->QTD - 2 ; AJUDA >= 0 ; AJUDA--)
      {
      P->NO[AJUDA].ELAPSED++;
      }
    for (AJUDA = E->QTD - 1 ; AJUDA >= 0 ; AJUDA--)
      {
      E->NO[AJUDA].ELAPSED++;
      E->NO[AJUDA].REALIO--;
      }
    }
    CONTA++;
```

```
    }

    /* Se nao implementa IO */
    if (CORRETO == 1)
    {
      P->NO[P->QTD-1].REALTIME++;
      for (AJUDA = P->QTD - 2 ; AJUDA >= 0 ; AJUDA--)
      {
      P->NO[AJUDA].ELAPSED++;
      }
      for (AJUDA = E->QTD - 1 ; AJUDA >= 0 ; AJUDA--)
      {
      E->NO[AJUDA].ELAPSED++;
      E->NO[AJUDA].REALIO--;
      }
    }
  }
}

void VERIFICAIO (Tlista *P, Tlista *E)
{
  int CONTA;
  Tprocesso TEMP;
  CONTA = 0;

  if (LISTAVAZIA (E) == 0)
  {
    while (CONTA < E->QTD)
    {
      /* Verifica termino de IO */
      if (E->NO[CONTA].REALIO == 0)
      {
        TEMP = E->NO[CONTA];
      E->NO[CONTA] = E->NO[E->QTD - 1];
      E->QTD--;
```

```
        INSERENO (P, TEMP);
        }
        CONTA++;
      }
    }
  }

/* ****************************************** */
/*      Mostra informacoes do processo       */
/* ****************************************** */
void DISPLAY (Tlista *P, Tlista *E, int CLOCK)
{
  int CONTA;

  clrscr();
  printf (   \n    Politica de escalonamento: PRIORIDADE   ;
  printf (              Unidade Tempo: %d\n\n\n   ,CLOCK);

  if (!LISTAVAZIA (P))
  {
    printf (     Processo PID: %d   ,P  ->NO[P->QTD-1].PID);
    printf (    Estado: EXECUC   ;
    printf (    Prio:  %d   ,P ->NO[P->QTD-1].PRIO);
    printf (    CPU -Time: %d   ,P ->NO[P->QTD-1].REALTIME);
    printf (    Elapsed -Time: %d\n   ,P ->NO[P->QTD-1].ELAPSED);

    for (CONTA = (P->QTD - 2) ; CONTA >= 0  ; CONTA--)
    {
      printf (     Processo PID: %d   ,P  ->NO[CONTA].PID);
      printf (    Estado: PRONTO   ;
      printf (    Prio: %d   ,P  ->NO[CONTA].PRIO);
      printf (    CPU -Time: %d   ,P ->NO[CONTA].REALTIME);
      printf (    Elapsed -Time: %d\n   ,P ->NO[CONTA].ELAPSED);
    }
  }
```

```
  if (!LISTAVAZIA (E))
  {
    for (CONTA = 0 ; CONTA < E->QTD ; CONTA++)
    {
      printf (     Processo PID: %d   ,E  ->NO[CONTA].PID);
      printf (   Estado: ESPERA   ;
      printf (   Prio: %d   ,E  ->NO[CONTA].PRIO);
      printf (   CPU-Time: %d   ,E ->NO[CONTA].REALTIME);
      printf (   Elapsed-Time: %d\n   ,E ->NO[CONTA].ELAPSED);
    }
  }

  printf (   \n\n\n          Pressione ENTER para seguir processamento   ;
  getchar ();
}

/* ****************************************** */
/*              Programa principal                          */
/* ****************************************** */
void main (void)
{
  int SAI, CONTA, RES, CLOCK, FLAG;
  int PIDDIS[12];
  Tlista PRONTO, ESPERA;
  Tprocesso NOVO;

  SAI = 0;
  RES = 1;
  FLAG = 0;
  INICIAPID (&PIDDIS);
  INICIALISTA (&PRONTO);
  INICIALISTA (&ESPERA);

  clrscr();
```

```
while (SAI != 5)
{
  printf (   \n\n   Escalonador de Processos   ;
  printf (   \n\n   1 - Definir processos   ;
  printf (   \n   2 - Escalonar por prioridade   ;
  printf (   \n   3     Opcao Inativa!   ;
  printf (   \n   4 - Versao do programa   ;
  printf (   \n\n   5 - Sair do escalonador   ;
  printf (   \n\nDigite a opcao desejada:    ;
  scanf (    d    ,&SAI);
  clrscr();

  if (SAI==1)
  {
    FLAG = 1;
    NOVO.PID = 1;
    while ((NOVO.PID != 0) && (RES != 0))
    {
    printf (    \n\n    PIDs disponiveis:    ;
      CONTA = 0;
    while (CONTA < 12)
     {
      if (PIDDIS[CONTA] == 1)
       {
        printf (    d     ,PIDDIS[CONTA+1]);
        }
      CONTA = CONTA + 2;
      }
    printf (    \n\n    Entre o PID desejado ou 0 para encerrar.    ;
    scanf (    d    ,&NOVO.PID);
    clrscr();

    if (((NOVO.PID<100) || (NOVO.PID>105)) && (NOVO.PID != 0))
     {
```

```
        printf (   \n\n    +++++ %d - PID invalido +++++   ,NOVO.PID);
  }


 CONTA = 1;
if ((NOVO.PID > 99) && (NOVO.PID < 106))
{
while (CONTA < 12)
  {
  if ((PIDDIS[CONTA]==NOVO.PID)&&(PIDDIS[CONTA-1]==0))
  {
     printf (   \n\n    +++++ %d - PID duplicado +++++   ,NOVO.PID);
  }


  if ((PIDDIS[CONTA]==NOVO.PID)&&(PIDDIS[CONTA-1]==1))
  {
  PIDDIS[CONTA-1] = 0;
  CRIAPROCESSO (&NOVO);
  ORDENAES (&NOVO);
  RES = INSERENO (&PRONTO, NOVO);
  if (RES == 0)
    {
    printf (   \n\n    Processo nao inserido !   ;
    printf (   \n\n    Nao ha capacidade para mais processos !   ;
    }
 if (RES == 1)
    {
    printf (   \n\n    Processo inserido com sucesso !   ;
    }
 printf (       \n    Pressione qualquer tecla para continuar.   ;
 getch();
 clrscr();
  }


 CONTA = CONTA + 2;
}
```

```
      }
    }

    clrscr();
  }

  if ((SAI==2) && (FLAG==1))
  {
    CLOCK = 0;
    while ((!LISTAVAZIA (&PRONTO)) || (!LISTAVAZIA (&ESPERA)))
    {
      DISPLAY (&PRONTO, &ESPERA, CLOCK);
      ORGANIZAPRIO (&PRONTO);
      PROCESSADOR (&PRONTO, &ESPERA, &CLOCK);
      VERIFICAIO (&PRONTO, &ESPERA);
      CLOCK++;
    }

    printf (   \n\n    Processamento terminado.   ;
    return;
  }

  if (SAI==4)
  {
    clrscr();
    printf (   \n\n\n   ;
    printf (                              Prof. Francis\n\n   ;
    printf (                      Carlos Henrique Pereira de Oliveira\n   ;
    printf (                             kkrj@terra.com.br\n   ;
    printf (    \n\n Resumo do trabalho:\n\n   ;
    printf (      O programa simula um escalonador por prioridades.\n   ;
    getch();
    clrscr();
   }
  }
```

```
}
```

Estudando com cuidado esse programa, observa-se que um processo pode se destacar com relação a outro, alterando seu estado de prioridade por haver uma necessidade de um tempo maior em unidades de tempo do uso da UCP.

entra f050202.jpg

Figura 5.2.2 – Após o escalonamento, os processos mudam de ordem.

5.2.3 Múltiplas Filas

Nesse tipo de escalonamento, os processos entram em filas conforme determinada característica que possuam em comum. Imaginemos processos que interagem com o usuário, processos do sistema operacional e processos em lote. São três filas distintas com suas respectivas prioridades obedecendo a uma ordem para serem executadas.

A fila de prioridade máxima seria a de processos de sistema operacional, seguida pela de processos do usuário e por último os processamentos em lote. Quando um processo é chamado, ele entra ao final de sua respectiva fila, e sua fila só entra em execução quando a fila anterior a ele estiver completamente vazia.

```
#include <stdio.h>


void main()
{
  int i=1,a, b, c=0, x, fila1[10], fila2[10], fila3[10], f1=0, f2=0, f3=0, tipo;


  while (i == 1)
  {
    printf(典ipo de processo: (1) S.O. (2) Usuario (3) Lote ou (0) Sair:    ;
    scanf(   i      &tipo);


    if (tipo == 0)
    {
      i = 0;
      break;
    }
    else
    {
      if (tipo == 1)
      {
```

```
      printf("Digite um numero para esse processo %i, ou 0 para fim:    f1);
      scanf(   i   &fila1[f1]);
      f1++;
     }

    if (tipo == 2)
    {
      printf("Digite um numero para esse processo %i, ou 0 para fim:    f2);
      scanf(   i   &fila2[f2]);
      f2++;
     }

    if (tipo == 3)
    {
      printf("Digite um numero para esse processo %i, ou 0 para fim:    f3);
      scanf(   i   &fila3[f3]);
      f3++;
     }

  }

  c++;

}

if (c > 0)
{
  for(a=0; a<c; a++)
  {
    if (a == 0)
      x = f1;
    if (a == 1)
      x = f2;
    if (a == 2)
      x = f3;
```

```
      for(b=0; b<x; b++)
      {
        if (a == 0)
          printf(   \nFila 1-Processo em execucao: %i   fila1[b]);
        if (a == 1)
          printf(   \nFila 2-Processo em execucao: %i   fila2[b]);
        if (a == 2)
          printf(   \nFila 3-Processo em execucao: %i   fila3[b]);
      }
    }
  }
  else
    printf(暫ao existem processos para execucao.   ;

}
```

Neste exemplo, as variáveis *f1*, *f2*, *f3* representam os contadores, no caso temos três filas de prioridade, ou seja, um contador para cada uma. Temos três filas (representadas por *fila1*, *fila2* e *fila3*), sendo uma para cada tipo de processo (S.O., usuários, lote). Quando a fila prioritária entra em execução, as outras duas ficam aguardando até que entrem em execução em ordem de prioridade.

entra f050203.jpg

Figura 5.2.3 – Processos são organizados por tipo e por prioridade.

5.2.4 Tempo Real

Quando é feito uso do escalonamento em tempo real, toda decisão tomada é com base na prioridade (que é estática) de cada processo, baseando-se no fato que o tempo é um fator de extrema importância.

Para se ter uma idéia clara disto, basta imaginar uma montadora de automóveis, onde os processos não podem falhar e precisam estar dentro de espaços precisos de tempo para que não ocorram problemas.

```
#include <stdio.h>
#include <conio.h>

void ordena();
int c, prio[100];
```

```
char proc[100];


void main()
{
  int i=1, b;


  while (i == 1)
  {
    printf(    Digite um caractere p/ esse processo %i, ou 0 para finalizar:    c);
    proc[c]=getche();


    if (proc[c] ==
    {
      i = 0;
      break;
    }
    else
    {
      printf(    \nDigite a prioridade desse processo:    ;
      scanf(   i   &prio[c]);
    }


    ordena();


    c++;


  }


  if (c > 0)
  {
    for(b=0; b<c; b++)
      printf(    \nOrdem final dos processos: %c    proc[b]);
  }
  else
    printf(  ao existem processos para execucao.   ;
```

```
}


void ordena()
{
  int t1, a;
  char t2[1];


  if (c > 0);
  {
    for(a=c; a>0; a--)
    {
      if (c > 0)
      {
        if (prio[a] < prio[a-1])
        {
          getch();
          t1 = prio[a];
          t2[0] = proc[a];
          prio[a] = prio[a-1];
          proc[a] = proc[a-1];
          prio[a-1] = t1;
          proc[a-1] = t2[0];
        }
      }
    }
  }
}
```

Note-se que cada vez que um processo entra na fila (representada pelo vetor *proc*), é vista sua prioridade (vetor *prio*) com as dos processos anteriores, se ele tiver uma prioridade de valor menor, ou seja, tem maior prioridade, ele "entra" na frente na fila de execução. Esse processo de alternar a posição é feito pela rotina *ordena*, que usando espaços temporários (*t1* e *t2*), faz a troca de posições do nome do processo e da prioridade do mesmo.

entra f050204.jpg

Figura 5.2.4 – Prioridade estática muda a ordem de execução.

5.2.5 Múltiplos Processadores

Quando é feito uso do escalonamento em tempo real, toda decisão tomada é com base na prioridade, que é estática, de cada processo, baseando-se no fato que o tempo é um fator de extrema importância.

Para se ter uma idéia clara disto, basta imagina uma montadora de automóveis, aonde os processos não podem falhar e precisão estar dentro de espaços precisos de tempo para que não ocorram problemas.

```
#include <stdio.h>
void main()
{
  int i=1, a, c=0, n, ponteiro, z;
  char proc[100], ucp1[100], ucp2[100], ucp3[100];
  int ucp1f=0, ucp2f=0, ucp3f=0;

  while (i == 1)
  {
    printf(   \nDigite um caractere p/ esse processo %i,ou 0 para finalizar:    c);
    proc[c]=getche();
    z=0;

    if (proc[c] ==
    {
      i = 0;
      break;
    }

    if (ucp1f == 0)
    {
      ucp1f = 1;
      ucp1[0] = proc[c];
      z=1;
    }
```

```
  if (z == 0)
  {
    if (ucp2f == 0)
    {
      ucp2f = 1;
      ucp2[0] = proc[c];
      z=1;
    }
  }

  if (z == 0)
  {
    if (ucp3f == 0)
    {
      ucp3f = 1;
      ucp3[0] = proc[c];
    }
  }

  c++;

}

i = 1;
ponteiro = 3;

if (c > 0)
{

  while (i == 1)
  {
     printf(   \nProcesso na UCP1: %c   ucp1[0]);
     printf(   \nProcesso na UCP2: %c   ucp2[0]);
     printf(   \nProcesso na UCP3: %c   ucp3[0]);
```

```
    for(a=ponteiro; a<c; a++)
    {
      printf(   \nProcesso na fila por aguardo de UCP livre: %c    proc[a]);
    }

    if (ponteiro == c)
    {
      i = 0;
      break;
    }

    printf(   \nDigite o numero da UCP para liberar:    ;
    scanf(   i    &n);

    if (n == 1)
    {
      ucp1[0] = proc[ponteiro];
    }
    if (n == 2)
    {
      ucp2[0] = proc[ponteiro];
    }
    if (n == 3)
    {
      ucp3[0] = proc[ponteiro];
    }

    ponteiro++;

  }

}
else
  printf(暂ao existem processos para execucao.   ;
```

```
}
```

Nesse programa temos um fila de processos principal (vetor proc), e outras três filas.

Veremos mais adiante em nosso próprio sistema operacional a implementação prática do FIFO, que é um tanto quanto simples e fácil de ser codificado por um programador. De momento veremos um exemplo em linguagem C.

Exercícios do Capítulo 5

1. Defina um escalonador do tipo FIFO e demonstre seus uso com os processos A, B e C, chamados nessa mesma ordem.

2. Qual o principal problema nos sistemas do tipo Cooperativo?

3. Como os processos que se organizam nos sistemas de Múltiplas Filas? Exemplifique a situação.

4. Defina e diferencie sistemas não-preemptivos de preemptivos.

5. Qual a característica do sistema de Tempo Real que define seu escalonamento e qual situação deve apresentar esse característica?