UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA
CURSO DE CIÊNCIAS DA COMPUTAÇÃO

Trabalho Prático 1: Escalonamento de Processos

Nelson Luiz Joppi Filho (17100527)
Matheus Henrique Schaly (18200436)

- **Como funciona o gerador de números aleatórios implementado na solução?**

O gerador de números aleatórios faz a seguinte conta: "*max_tickets ? krand() % max_tickets : 0*", *max_tickets* é um número que indica a quantidade máxima de *tickets* se somarmos todos *tickets* de todos processos, *krand()* é uma função já existente do *Nanvix* que gera um número inteiro grande pseudo-aleatório. O que acontece é que queremos usar a aleatoriedade do *krand()* e limitá-lo para no máximo ser *max_tickets* - 1, e isso é feito através do módulo. A operação ternária existe para que não façamos módulo com o *max_tickets* tendo valor de 0 (o que resultaria em um erro), e caso isso aconteça, o valor atribuído ao número sorteado é 0.

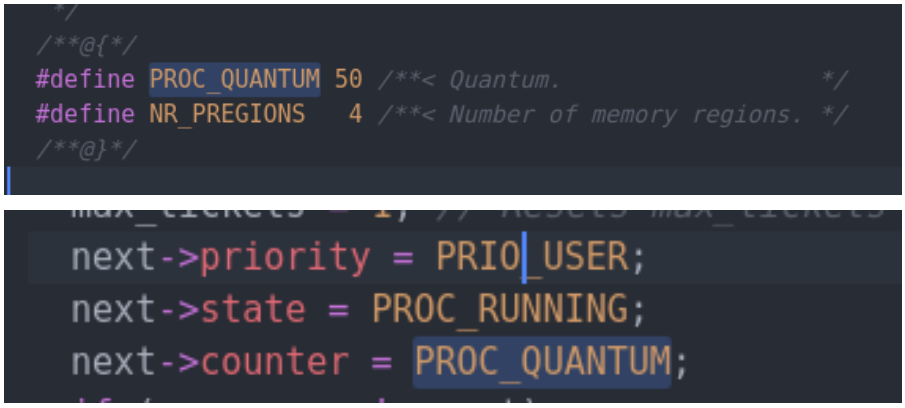
```
/*
    Counts the number of total tickets so that we
    can generate a random number between 0 and max_tickets - 1
*/
unsigned generate_rand_number()
{
  return max_tickets ? krand() % max_tickets : 0;
}
```

```
selected_ticket = generate_rand_number();
```

- **Qual o tamanho de *quantum* foi escolhido? Por quê?**

O tamanho do *quantum* é de 50. Optou-se por manter o mesmo *quantum* do próprio sistema, dado pela constante *PROC_QUANTUM* definida no arquivo *pm.h*. Fez-se isso pois acredita-se que o SO do *Nanvix* foi elaborado a fim de ter um equilíbrio entre os processos. Curto o suficiente para dar a ilusão de execução simultânea dos processos, mas longa o suficiente para que o tempo gasto pelo *Nanvix*

não seja muito significativo quando comparado ao tempo de execução de fato do processo.



- **Qual foi a estratégia de distribuição de tickets entre os processos? O que acontece quando um novo processo é criado e terminado?**

A distribuição de tickets entre os processos foi atribuída considerando a prioridade do processo. Quanto maior a prioridade, menor o valor do inteiro, e maior a quantidade de tickets. Por exemplo, a prioridade máxima é dada à operação de bloqueio, *PRIO_IO*, com valor de -100. Portanto, o processo que possui *PRIO_IO* como prioridade receberá a maior quantidade de *tickets* (16) em relação aos processos que possuem outras prioridades. Por outro lado, a prioridade mínima é dada à operação de usuário, *PRIO_USER*, com valor de 40. Portanto, o processo que possui PRIO_USER como prioridade receberá a menor quantidade de *tickets* (2) em relação aos processos que possuem outras prioridades. Porém, caso nenhum processo esteja pronto para ser executado, nenhum processo será sorteado, logo quem irá ser executado ser o processo *IDLE*.

Em relação aos *tickets*, quando um processo é criado, no arquivo *fork.c*, seu número de tickets é setado para 0. Quando o processo é levado para ser agendado, na função *sched*, no arquivo *sched.c*, atribui-se *PROC_READY* ao atributo state do processo, seta-se o número de *tickets* do processo baseado em sua prioridade, atribui-se os *compensation tickets*, e zera-se o counter. E em relação aos *tickets*, quando um processo é terminado, o atributo *tickets* deixa de existir, juntamente com o processo.

Em relação a criação e termino de processo no Nanvix temos para criação: “Para criar um processo, o kernel primeiro procura por um slot vazio na *processtable*, para armazenar ali as informações sobre o processo. Em seguida, um espaço de endereço para o processo é criado: as tabelas de página são inicializadas, o código do kernel e os segmentos de dados são anexados ao núcleo do processo e uma pilha de kernel para o processo é criada. Depois disso, o kernel duplica todas as regiões de memória do processo pai e às anexa ao espaço de endereço do processo filho.” E temos para término: “Quando o processo finalmente conclui seu trabalho, ele invoca a

chamada do sistemaexit() para terminar. Essa chamada ordena que o subsistema de gerenciamento de processo realmente interrompa o processo e libere todos os recursos atribuídos a ele.”

```
/**
 * @name Process priorities
 */
/**@{*/
#define PRIO_IO         -100 /**< Waiting for block operation. */
#define PRIO_BUFFER      -80 /**< Waiting for buffer.          */
#define PRIO_INODE       -60 /**< Waiting for inode.           */
#define PRIO_SUPERBLOCK  -40 /**< Waiting for super block.     */
#define PRIO_REGION      -20 /**< Waiting for memory region.   */
#define PRIO_TTY           0 /**< Waiting for terminal I/O.    */
#define PRIO_SIG          20 /**< Waiting for signal.          */
#define PRIO_USER         40 /**< User priority.               */
/**@}*/
```

```c
PUBLIC void sched(struct process *proc)
{
  proc->state = PROC_READY;

  /* Grant a ticket to the process based on priority */
  if (proc->priority == PRIO_IO) {
    proc->tickets = 16;
  } else if (proc->priority == PRIO_BUFFER) {
    proc->tickets = 14;
  } else if (proc->priority == PRIO_INODE) {
    proc->tickets = 12;
  } else if (proc->priority == PRIO_SUPERBLOCK) {
    proc->tickets = 10;
  } else if (proc->priority == PRIO_REGION) {
    proc->tickets = 8;
  } else if (proc->priority == PRIO_TTY) {
    proc->tickets = 6;
  } else if (proc->priority == PRIO_SIG) {
    proc->tickets = 4;
  } else {
    proc->tickets = 2;
  }

  /*
    Because counter is used as a clock tick counter when the process
    is running, we can use it to calculate its quantum_fraction
    quantum_fraction is how many ticks were executed
    The compensation is given by PROC_QUNATUM / proc->quantum_fraction
  */
  proc->quantum_fraction = PROC_QUANTUM - proc->counter;
  /* Grant compensation tickets */
  if (proc->quantum_fraction != 0)
    proc->tickets = proc->tickets * PROC_QUANTUM / proc->quantum_fraction;
  proc->counter = 0;
}
```

```c
/**
 * @brief Stops the current running process.
 */
PUBLIC void stop(void)
{
  curr_proc->state = PROC_STOPPED;
  sndsig(curr_proc->father, SIGCHLD);
  yield();
}
```

- **Qual foi a estratégia utilizada para determinar o processo ganhador quando um ticket aleatório é sorteado?**

  Usa-se a ideia descrita no artigo para encontrar o processo sorteado.

  A variável *max_tickets* é a quantidade total de *tickets* de todos os processos do sistema que estão no estado *PROC_READY*. Tal variável é computada quando a

função de escalonamento *yield* é invocada. Percorre-se todos os processos armazenando a soma de seus *tickets* na variável *max_tickets*. Em seguida, invoca-se o gerador de número aleatórios *generate_rand_number*, armazenando-se o valor em *selected_ticket*.

```
/*
  Count the number of total tickets so that we
  can generate a random number between 1 and max_tickets
*/
for (p = FIRST_PROC; p <= LAST_PROC; p++)
{
  if (p->state != PROC_READY)
    continue;

  max_tickets += p->tickets;
}

selected_ticket = generate_rand_number();
```

Em seguida temos mais um *for loop* que percorre todos os processos que estão no estado *PROC_READY*. Para cada estado percorrido, usa-se a variável *ticket_sum* para realizar a soma de tickets ocorridas até o momento. Quando a variável *ticket_sum* for maior que a variável *selected_ticket* isso significa que encontramos o *ticket* sorteado e consequentemente o processo sorteado.

```
/* Choose a process to run next. */
next = IDLE;
for (p = FIRST_PROC; p <= LAST_PROC; p++)
{
  /* Skip non-ready process. */
  if (p->state != PROC_READY)
    continue;

  ticket_sum += p->tickets;

  /*
   * Finds the process with
   * the winning ticket
   */
  if (ticket_sum > selected_ticket)
  {
    next = p;
    break;
  }
}
```

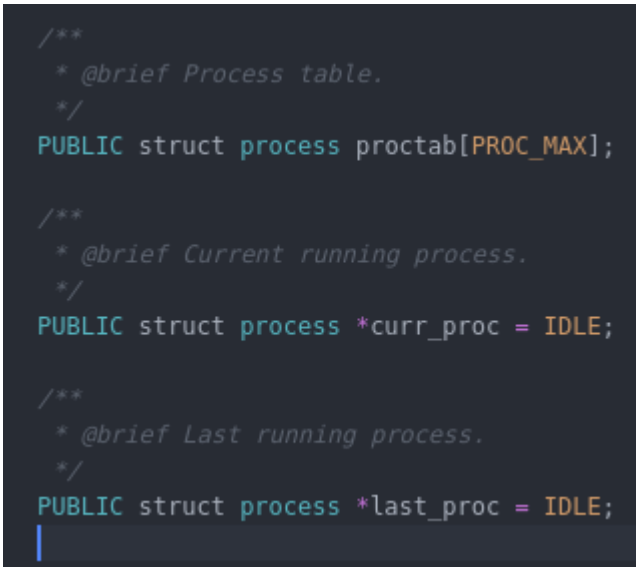
- **Como você implementou a ideia de compensation tickets?**
  Utilizou-se da ideia descrita no artigo.

Quando um processo passa a ser agendado, na função *sched,* do arquivo *sched.c*, calculamos a *quantum_fraction* que é a fração dos *ticks* do processo que foram executadas, dado por *PROC_QUANTUM - counter*. Em seguida, calculamos os *tickets* ao multiplicar a quantidade de *tickets* do processo com a sua compensação. A compensação é dada pela divisão do total que o processo deveria ter executado *PROC_QUANTUM* dividido pelo *quantum_fraction*.

Por exemplo: dado um *PROC_QUANTUM* de 100, dado que o processo finalizou sua execução com um *counter* de 80 e que ele possui 400 *tickets* temos: *quantum_fraction* = 100 - 80 = 20, ou seja 20 *ticks* foram executados. A compensação é dada por *PROC_QUANTUM / quantum_fraction*, ou seja, 100/20 = 5. Sendo assim, temos que os *tickets* junto com a compensação é dado por *tickets * compensation* = 400 * 5 = 2000.

- **Foram utilizadas estruturas para implementar a sua solução? Quais?**

Não. Nenhuma estrutura de dados adicional foi utilizada. Percorre-se apenas a tabela *proctab* localizada em *pm.c*. Tal tabela possui todos os 64 possíveis processos do sistema.

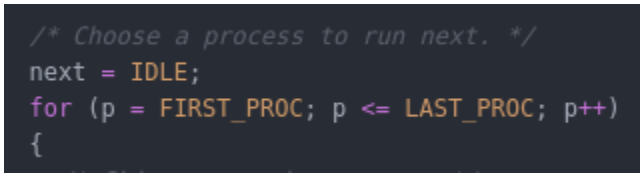


```
/**
 * @brief Process table.
 */
PUBLIC struct process proctab[PROC_MAX];

/**
 * @brief Current running process.
 */
PUBLIC struct process *curr_proc = IDLE;

/**
 * @brief Last running process.
 */
PUBLIC struct process *last_proc = IDLE;
```

Percorre-se, na função *sched.c*, do arquivo *sched*, iniciando do processo do índice 1 (índice 0 é o processo *IDLE)* até o último processo do sistema, similar a uma estrutura de fila.



```
/* Choose a process to run next. */
next = IDLE;
for (p = FIRST_PROC; p <= LAST_PROC; p++)
{
```