Ana Clara Mueller Miranda - RA 148806
Bruno Pires Moreira Silva - RA 139892
Igor Ribeiro Ferreira de Matos - RA 140492
Rafaela Cristine dos Santos Uchôas - RA 140351

Relatório Projeto 3 - Escalonamento

1. Sobre o Projeto e o Minix

O projeto consiste numa implementação dos algoritmos de escalonamento para sistemas interativos no sistema operacional Minix 3, com o objetivo de observar o comportamento distinto do escalonador e o impacto que isto causa no tempo de espera entre os processos.

O Minix 3 é um sistema operacional open source e microkernel, com algoritmos de escalonamento de Múltiplas Filas de Prioridade (em kernel) e Round Robin (em espaço de usuário, rodando para cada fila).

Round Robin:

É um algoritmo preemptivo que garante que não ocorrerá inanição entre os processos com uma porção de tempo de execução fixo distribuído entre eles chamado de “quantum”.

Múltiplas Filas de Prioridade:

A ideia principal é manter 16 filas de diferentes prioridades distintas, onde os processos de maiores prioridades são executados primeiro. A fim de tratar os processos que têm a mesma prioridade, é chamado o escalonador round robin que trata os processos de cada fila, e vai diminuindo suas prioridades a cada chamada, para evitar que um (ou um grupo de) processo domine a CPU indeterminadamente.

2. Implementações

a) Escalonamento de prioridade

Com um funcionamento bastante semelhante ao já implementado, o escalonamento escolhe uma série de processos com diferentes prioridades e executa primeiro os que têm mais prioridade, com a possibilidade de inanição nos processos com baixa prioridade.
Quando há uma necessidade para escalonamento, o main.c de ../sched/* é inicializado e, com os dados recebidos,é feita uma chamada de SEF (System Event Framework), uma biblioteca de chamadas para auxiliar a inicialização de um forma fácil e efetiva. Aqui na main é onde haverá o loop principal da /Sched/.

```
int main(void)
{
        /* Main routine of the scheduler. */
        message m_in;   /* the incoming message itself is kept here. */
        int call_nr;    /* system call number */
        int who_e;      /* caller's endpoint */
        int result;     /* result to system call */
        int rv;

        /* SEF local startup. */
        sef_local_startup();

        /* This is SCHED's main loop - get work and do it, forever and forever. */
        while (TRUE) {
```

Então, a partir das chamadas sef, init_scheduling da /sched/schedule.c é chamada é inicializada, onde a função fica sempre aguardando por mensagens.

```c
/*===========================================================================*
 *                            init_scheduling                                *
 *===========================================================================*/
void init_scheduling(void)
{
        int r;

        balance_timeout = BALANCE_TIMEOUT * sys_hz();

        if ((r = sys_setalarm(balance_timeout, 0)) != OK)
                panic("sys_setalarm failed: %d", r);
}
```

Então, sempre que SCHED receber uma mensagem para controlar algum processo, o laço principal da main.c recebe e decide chamar “do_start_scheduling”, “do_stop_scheduling”, “do_nice” ou “do_no_quantum”

```c
switch(call_nr) {
case SCHEDULING_INHERIT:
case SCHEDULING_START:
        result = do_start_scheduling(&m_in);
        break;
case SCHEDULING_STOP:
        result = do_stop_scheduling(&m_in);
        break;
case SCHEDULING_SET_NICE:
        result = do_nice(&m_in);
        break;
case SCHEDULING_NO_QUANTUM:
        /* This message was sent from the kernel, don't reply */
        if (IPC_STATUS_FLAGS_TEST(ipc_status,
                IPC_FLG_MSG_FROM_KERNEL)) {
                if ((rv = do_noquantum(&m_in)) != (OK)) {
                        printf("SCHED: Warning, do_noquantum "
                                "failed with %d\n", rv);
                }
                continue; /* Don't reply */
        }
        else {
                printf("SCHED: process %d faked "
                        "SCHEDULING_NO_QUANTUM message!\n",
                                who_e);
                result = EPERM;
        }
        break;
default:
        result = no_sys(who_e, call_nr);
}

/* Send reply. */
if (result != SUSPEND) {
        m_in.m_type = result;           /* build reply message */
        reply(who_e, &m_in);            /* send it away */
}
```

E a cada chamada do scheduler, a prioridade do processo NÃO é rebaixada nas filas. Desta forma, o escalonamento irá executar todos os processos de máximo prioridade primeiro, para só depois ir executando os seguintes.

```c
/*===========================================================================*
 *              do_noquantum                 *
 *===========================================================================*/

int do_noquantum(message *m_ptr)
{
    register struct schedproc *rmp;
    int rv, proc_nr_n;

    if (sched_isokendpt(m_ptr->m_source, &proc_nr_n) != OK) {
        printf("SCHED: WARNING: got an invalid endpoint in OOQ msg %u.\n",
        m_ptr->m_source);
        return EBADEPT;
    }

    rmp = &schedproc[proc_nr_n];
    // if (rmp->priority < MIN_USER_Q) {
    //  rmp->priority += 1; /* lower priority */
    // }

    if ((rv = schedule_process_local(rmp)) != OK) {
        return rv;
    }
    return OK;
}
```

Ao desfazer os comentários das linhas responsáveis pelo rebaixamento da prioridade, o algoritmo voltaria a ter o funcionamento de múltiplas filas de prioridade.

A primeira comparação (Padrão do minix3 à esquerda e de prioridade a direita) podemos observar diferenças quase inexistentes, em um

./test 20 1000 100000

CPU	1	0	CPU	5	0
CPU	3	0	CPU	15	0
CPU	5	0	CPU	11	0
CPU	7	0	CPU	3	0
CPU	9	0	CPU	1	0
CPU	11	0	CPU	13	0
CPU	13	0	CPU	19	0
CPU	15	0	CPU	7	0
CPU	17	0	CPU	9	0
CPU	19	0	CPU	17	0
IO	0	4.06668	IO	0	4.06668
IO	2	4.08335	IO	6	4.06668
IO	4	4.08335	IO	4	4.06668
IO	6	4.1	IO	2	4.06668
IO	8	4.1	IO	8	4.08335
IO	10	4.1	IO	12	4.08335
IO	12	4.1	IO	10	4.08335
IO	14	4.1	IO	14	4.08335
IO	16	4.1	IO	16	4.08335
IO	18	4.1	IO	18	4.08335

Em um caso de teste maior, podemos observar que houve uma diferença de tempo de execução entre os processos, principalmente de entrada e saída, mostrando que o de prioridade foi um pouco menos eficaz para lidar com um número maior de E/S do que o padrão (múltiplas filas).

CPU	69	0	CPU	51	0
CPU	71	0	CPU	53	0
CPU	73	0	CPU	55	0
CPU	75	0	CPU	57	0
CPU	77	0	CPU	59	0
CPU	79	0	CPU	63	0
CPU	81	0	CPU	65	0
CPU	83	0	CPU	67	0
CPU	85	0	CPU	69	0
CPU	87	0	CPU	71	0
CPU	89	0	CPU	75	0
CPU	91	0	CPU	61	0.016675
CPU	93	0	CPU	73	0.016675
CPU	95	0	CPU	77	0
CPU	97	0	CPU	79	0
CPU	99	0	CPU	81	0
IO	6	2068.7	IO	82	2205.83
IO	20	2069.18	IO	26	2206.15
IO	2	2069.23	IO	6	2206.15
IO	12	2069.23	IO	92	2206.3
IO	38	2069.58	IO	90	2206.43
IO	54	2069.87	IO	10	2206.75
IO	8	2070.05	IO	36	2206.85
IO	66	2070.08	IO	12	2206.98
IO	80	2070.37	IO	2	2207
IO	68	2070.42	IO	72	2207
IO	36	2070.52	IO	4	2207.03
IO	72	2070.58	IO	18	2207.4
IO	98	2070.57	IO	16	2207.6
IO	58	2070.72	IO	68	2207.65
IO	78	2070.83	IO	84	2207.6
IO	14	2071.1	IO	32	2207.7
IO	96	2071	IO	86	2207.63
IO	30	2071.2	IO	60	2207.77
IO	60	2071.17	IO	44	2207.83
IO	34	2071.37	IO	20	2208.03
IO	22	2071.48	IO	80	2208.03
IO	52	2071.48	IO	56	2208.12
IO	56	2071.48	IO	64	2208.15
IO	24	2071.67	IO	74	2208.22
IO	0	2071.72	IO	0	2208.33
IO	4	2071.75	IO	88	2208.25
IO	40	2071.78	IO	30	2208.35
IO	82	2071.87	IO	14	2208.68
IO	64	2071.93	IO	42	2208.7
IO	10	2072.13	IO	94	2208.62
IO	18	2072.13	IO	98	2208.63
IO	28	2072.22	IO	96	2208.65
IO	86	2072.12	IO	38	2208.77
IO	90	2072.13	IO	24	2208.8
IO	26	2072.27	IO	8	2208.87
IO	32	2072.27	IO	22	2208.93

Por fim, podemos verificar que, em um teste de execução com um milhão de processos de entrada e saída, e cem milhões de processos de CPU, o algoritmo de prioridade se mostrou ser alguns segundos mais lento nos processos de entrada e saída.
(à esquerda, uma porção dos testes de execução do algoritmo padrão do minix, á direita, o escalonamento de prioridade)

CPU	85	0	CPU	69	0
CPU	87	0	CPU	71	0
CPU	89	0.01665	CPU	59	0.01665
CPU	91	0	CPU	63	0.016675
CPU	93	0	CPU	67	0.016675
CPU	95	0	CPU	73	0.01665
CPU	97	0	CPU	75	0
CPU	99	0	IO	78	27821.8
CPU	1	0	IO	80	27821.9
CPU	3	0	IO	82	27822
CPU	5	0	IO	84	27822
CPU	7	0	IO	86	27822.1
CPU	9	0	IO	88	27822.2
CPU	11	0	IO	90	27822.2
CPU	13	0	IO	92	27822.3
CPU	15	0	IO	94	27822.3
CPU	17	0	IO	96	27822.4
IO	38	27805.6	IO	98	27822.4
IO	32	27806.6	IO	0	27822.5
IO	66	27806.9	IO	2	27822.6
IO	12	27807.9	IO	4	27822.6
IO	76	27808.2	IO	6	27822.6
IO	52	27809.3	IO	8	27822.6
IO	96	27809.2	IO	10	27822.6
IO	98	27810.2	IO	12	27822.7
IO	10	27810.5	IO	14	27822.7
IO	70	27810.9	IO	16	27822.7
IO	58	27811.3	IO	18	27822.7
IO	22	27811.7	IO	20	27822.7
IO	34	27812.4	IO	22	27822.8
IO	50	27812.9	IO	24	27822.8
IO	86	27813.2	IO	26	27822.8
IO	40	27813.3	IO	28	27822.8
IO	36	27813.5	IO	30	27822.8
IO	0	27813.9	IO	32	27822.8
IO	48	27814	IO	34	27822.8
IO	2	27814.1	IO	36	27822.9
IO	24	27814.2	IO	38	27822.9
IO	62	27814.3	IO	40	27823
IO	28	27814.4	IO	42	27823
IO	64	27814.5	IO	44	27823
IO	26	27814.5	IO	46	27823
IO	46	27814.5	IO	48	27823
IO	82	27814.5	IO	50	27823
IO	90	27814.6	IO	52	27823
IO	14	27814.6	IO	54	27823
IO	72	27814.7	IO	64	27823

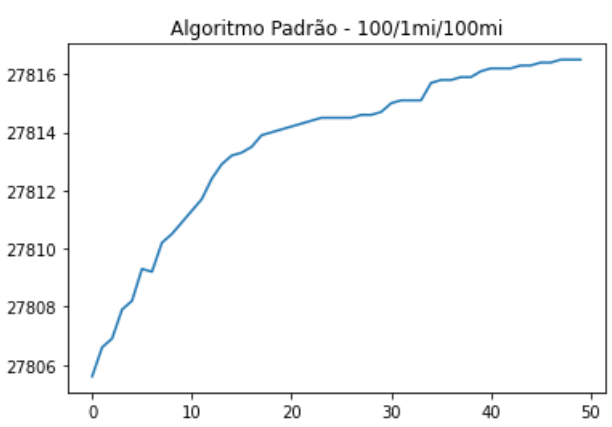
b) Já no segundo algoritmo de escalonamento, os processos são inseridos na fila de prioridade de forma ordenada, do maior processo ao menor processo. O arquivo em questão, que foi modificado proc.c, conforme a imagem abaixo.

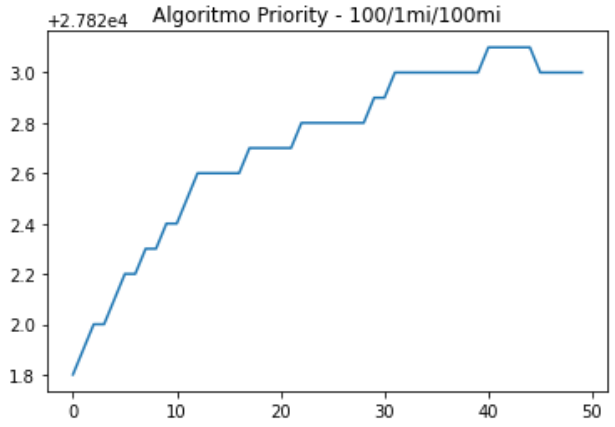
```
if (!rdy_head[q]) {       /* add to empty queue */
    rdy_head[q] = rdy_tail[q] = rp;       /* create a new queue */
    rp->p_nextready = NULL;       /* mark new end */
}
else {                    /* add to tail of queue */
      struct proc *aux= rdy_head[q];;     // Processo que ira iterar na fila

      //Itera na fila até o primeiro processo com tempo menor
      while(aux->p_nextready != NULL && rp->p_cpu_time_left > aux->p_nextready->p_cpu_time_left)
          aux = aux->p_nextready;

    rp->p_nextready = aux->p_nextready;//insere na fila
    aux->p_nextready = rp;
}
```

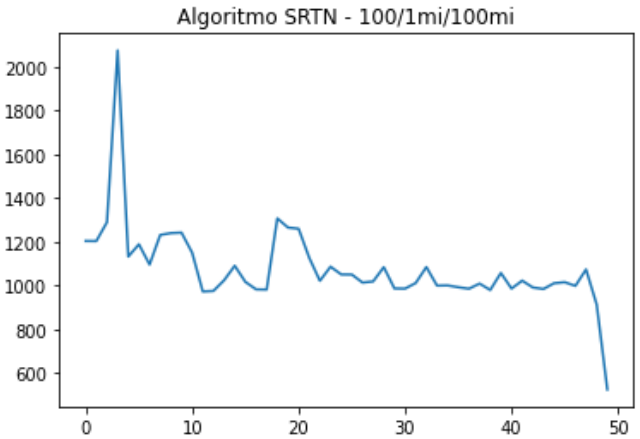
3. Comparação dos testes

Para fazer a comparação dos algoritmos nós plotamos gráficos considerando apenas processos IO bound sendo o eixo X o número de processos e Y o tempo decorrido. Os processos CPU bound não foram considerados pois o tempo era nulo em praticamente todos os casos.

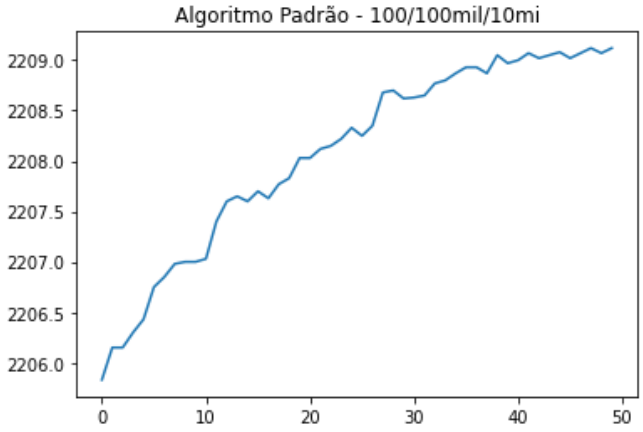
No primeiro teste consideramos um caso com 100 processos, 1 milhão de operações de IO por processo e 100 milhões de operações de CPU por processo.

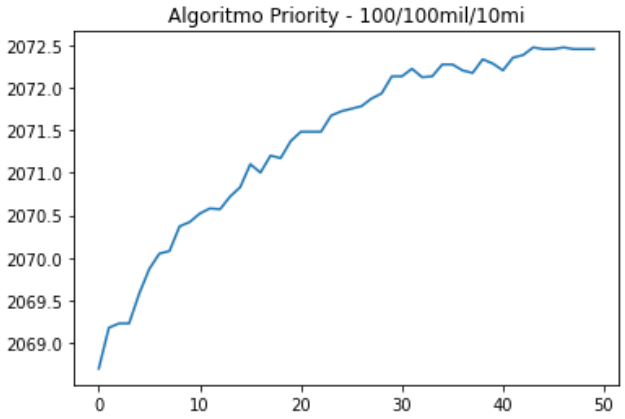


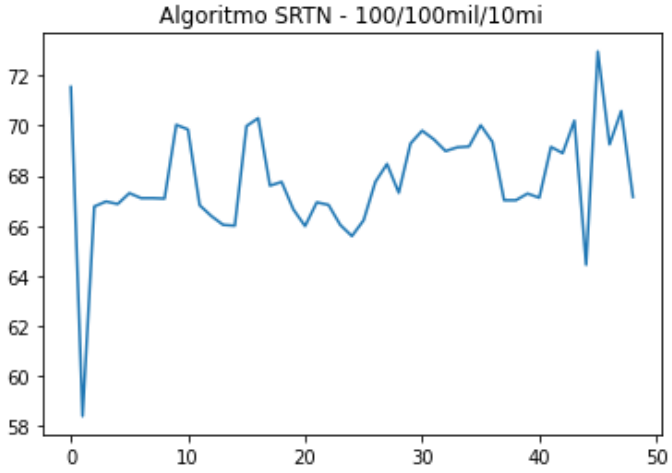




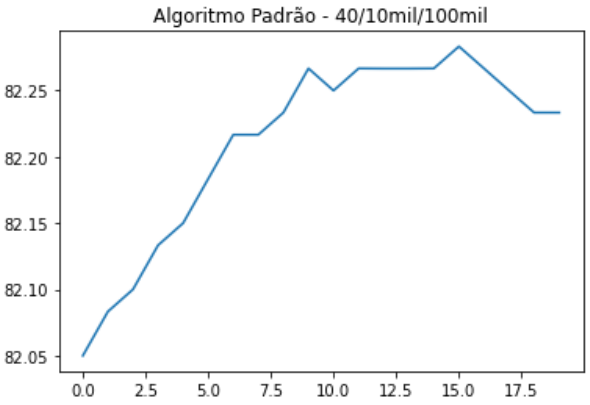
No segundo teste consideramos um caso com 100 processos, 100mil operações de IO por processo e 10 milhões de operações de CPU por processo.

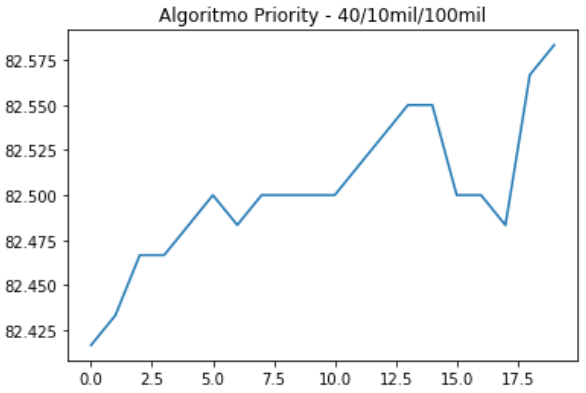


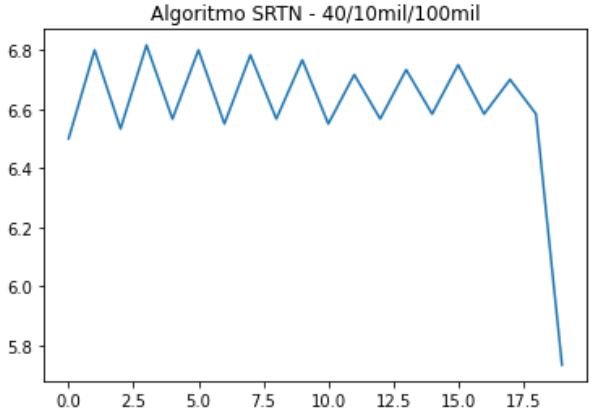




No terceiro e último teste consideramos um caso com 40 processos, 10 mil operações de IO por processo e 100 mil de operações de CPU por processo.

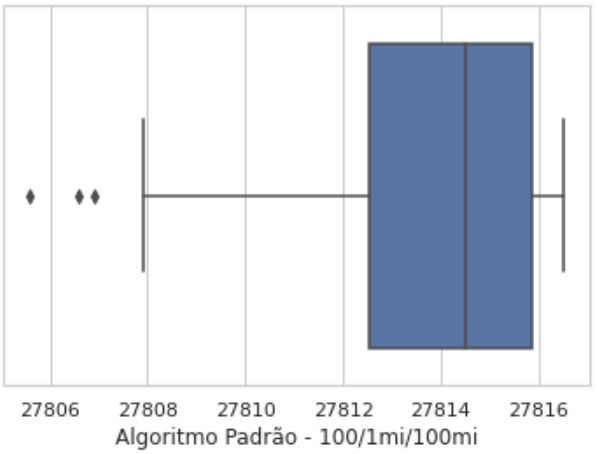


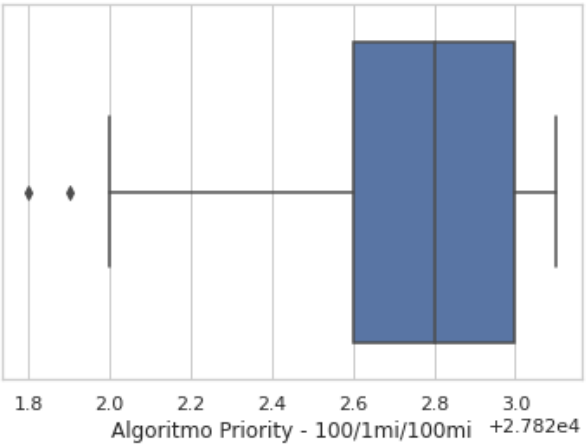


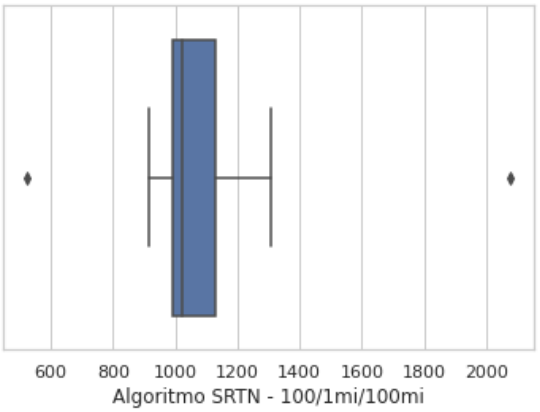


Além do gráfico de linhas, também podemos ter o boxplot de cada um dos tempos de cada um dos algoritmos:

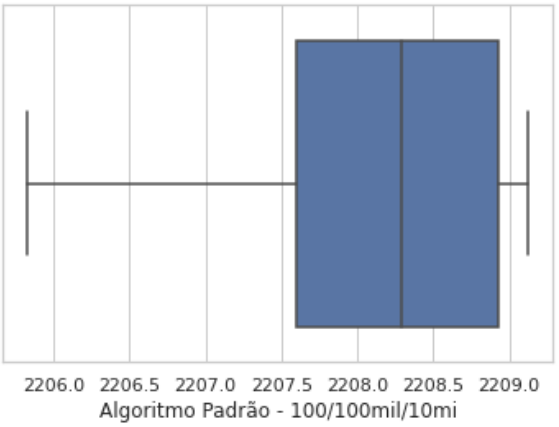
Boxplots do primeiro teste:

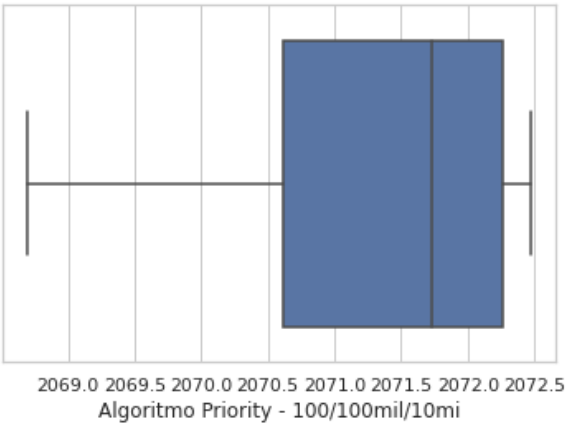


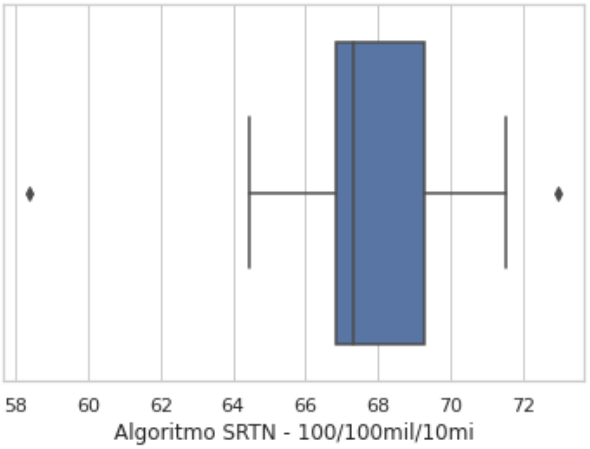




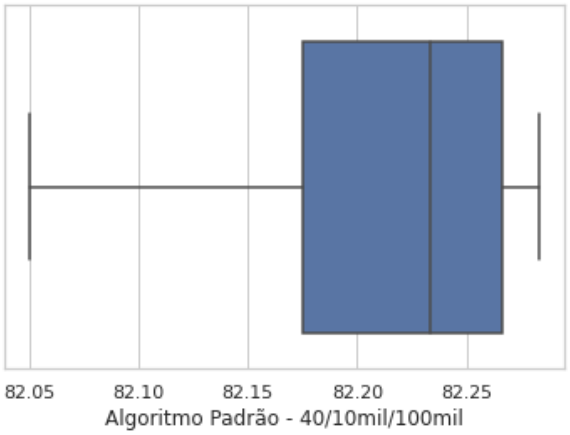
Boxplots do segundo teste:

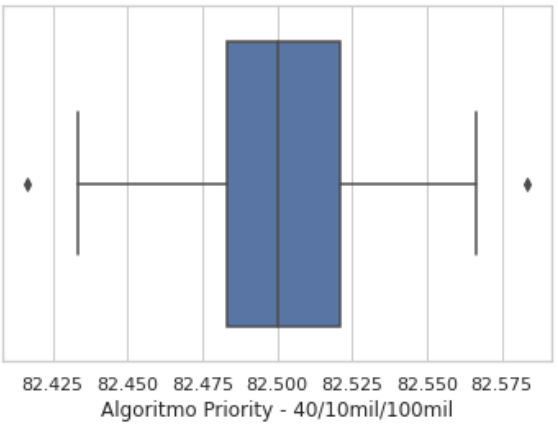


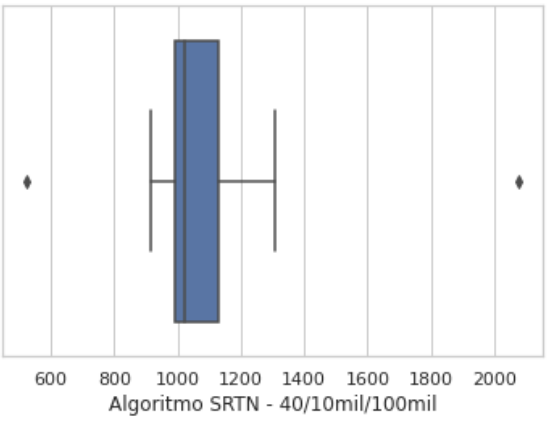




Boxplots do terceiro teste:







Analisando os três experimentos, o que podemos tirar de conclusão é que o algoritmo padrão é o mais estável para diferentes cenários, como ocorreu em nossas simulações. O crescimento de tempo ao longo dos processos ocorreu de forma mais próxima a linear possível, com uma certa estabilização à medida que novos processos eram instanciados. Além disso, a variação do tempo de cada processo ocorreu de forma mais ponderada entre todos os experimentos.

Já no caso do algoritmo priority, como houve uma dispersão significativa dos tempos gastos de cada processo dentro de um experimento, podemos afirmar que ele possui certa instabilidade em diferentes cenários, como fica evidente principalmente na primeira simulação, onde o boxplot torna mais clara a visualização desse comportamento. Também podemos perceber uma certa instabilidade, principalmente com os dados do terceiro experimento desse algoritmo, pois, embora os tempos dos diferentes processos estão em um intervalo bem definido, diferentemente do primeiro caso, não há um comportamento linear ao longo da instanciação dos novos processos.

Já o terceiro algoritmo, o SRTN, podemos ver a ocorrência de alguns outliers em todos os experimentos, o que não acontecia com os algoritmos anteriores. Entretanto, analisando a maioria dos casos, podemos ver como o tempo gasto nos casos gerais é próximo, como fica mais evidente nos boxplots. Porém, o comportamento que chama a atenção nesse algoritmo é a forma que os diferentes processos instanciados ao longo do tempo se comportam, sem padrões nas funções e com diversos processos novos que são executados em um intervalo mais curto, algo que era significativamente incomum nos outros algoritmos.

4. Organização do grupo

Após nos reunirmos para, primeiramente, instalar o minix nos computadores de todos os integrantes do grupo, a divisão dos membros que ficariam mais aplicados em cada código ficou da seguinte maneira:

Algoritmo de escalonamento do item a - Ana e Igor

Algoritmo de escalonamento do item b - Bruno e Rafaela

Sendo que não deixou de haver a discussão de todos os membros do grupo sobre aspectos de ambos os códigos, assim como no relatório.

5. Auto-avaliação do grupo

As principais dificuldades encontradas no decorrer do projeto certamente foram na instalação, configuração e manuseio do minix. Com isso solucionado, os experimentos foram executados com pequenos problemas, discutidos em aula com o professor. Conclui-se que todos colaboraram igualmente, de diferentes maneiras.