**RELATÓRIO DE LABORATÓRIO:**

**PING PONG OS P3 E P5**

**SISTEMAS OPERACIONAIS**

THIAGO MELCHER ARMÊNIO – 2358565

VICTOR AUGUSTO DEL MONEGO – 2378345

ENGENHARIA ELETRÔNICA – UTFPR

2 DE JULHO DE 2024

**1.Introdução Geral**

O seguinte relatório diz respeito aos laboratórios P3 e P5 de Sistemas Operacionais, enfatizando duas das aplicações da *PingPongOS*, do Prof Carlos Maziero.

Para a realização das análises a seguir, foram utilizados o CCS, aplicado em cima da placa TIVA C EK-TM4C1294XL, e a distro *Linux Mint 21.3*.

A proposta da prática P3 é utilizar um despachante, trabalhando em conjunto com um escalonador, para executar *threads* que se imprimem na tela em sequência.

**2.P3 - Despachante**

A figura 01 abaixo apresenta o código “*dispatcher.c*”, que é o código principal da prática.

|  |
| --- |
| #include "ppos.h"  **extern** **void** **dispatcher\_tiva** ();  **task\_t** Pang, Peng, Ping, Pong, Pung ;  // Corpo das threads  **void** **Body** (**void** \* arg)  {  **int** i ;  UARTprintf ("%s: inicio**\n**",(**char** \*)arg) ;  **for** (i=**0**; i<**5**; i++)  {  UARTprintf ("%s: %d**\n**", (**char** \*) arg, i) ;  task\_yield ();  }  UARTprintf ("%s: fim**\n**", (**char** \*) arg) ;  task\_exit (**0**) ;  }  **void** **dispatcher\_tiva** ()  {  UARTprintf ("dispatcher: inicio**\n**");  ppos\_init () ;  task\_create (&Pang, Body, " Pang") ;  task\_create (&Peng, Body, " Peng") ;  task\_create (&Ping, Body, " Ping") ;  task\_create (&Pong, Body, " Pong") ;  task\_create (&Pung, Body, " Pung") ;  task\_yield () ;  UARTprintf ("dispatcher: fim**\n**");  } |

Figura 1: código "dispatcher.c"

A figura 02 abaixo mostra o código “*ppos-core.c*”, que representa a maioria das funções ativamente responsáveis pelo escalonamento das *threads*.

|  |
| --- |
| #include "fila.h" // Filas  #include "microcontext.h" // Para troca de contexto  #include "ppos.h" // Declaracao de funcoes e defines do PPOS  #include "ppos\_d.h" // Estruturas de dados do Ping Pong  #define STACKSIZE 512  **task\_t** \*taskMain, \*taskAtual; // Task principal e atual  **typedef** **enum** idd {main, dispatcher} idd;  **int** taskID; // Contagem de ID de tasks  **task\_t** taskdisp; // Despachante  **task\_t** \*taskQueue; // Fila de tarefas  **task\_t** \*taskNext; // Próxima tarefa a ser executada  // Escalonamento de tasks  **task\_t** \***scheduler** () // Função chamada para escalonar as threads  {  **task\_t** \*taskSelection, \*taskAux;  taskSelection = taskQueue;  taskAux = taskQueue;  #ifdef DEBUG  UARTprintf("scheduler: proxima tarefa: %d - %s**\n**",  taskSelection->id, taskSelection->name);  #endif  **return** taskSelection;  }  **void** **Bodydispatcher** (**void** \* arg)  {  #**if** defined (DEBUG)|| defined(DEBUGlite)  UARTprintf("dispatcher: iniciado**\n**");  #endif  **while** (taskQueue){  taskNext = scheduler();  **if** (taskNext->status == PRONTA) {  queue\_remove ((**queue\_t** \*\*) &taskQueue, (**queue\_t**\*) taskNext);  task\_switch(taskNext);  }  }  task\_exit(**1**) ;  }  // Inicializa o sistema operacional; deve ser chamada no inicio do main()  **void** **ppos\_init** ()  {  setvbuf (stdout, **0**, \_IONBF, **0**);  taskID = **0**;  taskMain = (**task\_t**\*)malloc(**sizeof**(**task\_t**));  **char** \*stack;  stack = malloc (STACKSIZE);  **if**(stack)  {  taskMain->context.uc\_stack.ss\_sp = stack;  taskMain->context.uc\_stack.ss\_size = STACKSIZE;  taskMain->context.uc\_stack.ss\_flags = **0**;  taskMain->context.uc\_link = **0**;  taskMain->name = "main";  }  **else**  {  UARTprintf("Erro na criacao da pilha.**\n**");  exit(**1**);  }  taskMain->next = NULL;  taskMain->prev = NULL;  taskMain->id = taskID;  taskAtual = taskMain;  get\_context\_asm(&(taskMain->context));  makecontextMain (&(taskMain->context), **0**, **1**, "main");  #**if** defined (DEBUG)|| defined(DEBUGlite)  UARTprintf("ppos\_init: PPOS inicializado**\n**");  #endif  task\_create (&taskdisp,Bodydispatcher, "dispatcher"); // Criação do despachante  }  // Cria uma nova tarefa. Retorna um ID> 0 ou erro.  **int** **task\_create** (**task\_t** \*task, // Descritor da nova tarefa  **void** (\*start\_func)(**void** \*), // Funcao corpo da tarefa  **void** \*arg) // Argumentos para a tarefa  {  **char** \*stack; // Criacao da pilha para contexto  stack = malloc (STACKSIZE);  **if** (stack)  {  task->context.uc\_stack.ss\_sp = stack;  task->context.uc\_stack.ss\_size = STACKSIZE;  task->context.uc\_stack.ss\_flags = **0**;  task->context.uc\_link = **0**;  task->context.initialized = **0**;  task->status = PRONTA;  task->name = arg;  }  **else**  {  UARTprintf("Erro na criacao da pilha: ");  exit(**1**);  }  taskID++;  task->next = NULL;  task->prev = NULL;  task->id = taskID;  get\_context\_asm(&(task->context));  makecontext(&(task->context), (**int**)(\*start\_func), **1**, (**char**\*)arg);  **if** (task != &taskdisp) // Adiciona a tarefa criada ao final da fila  queue\_append ((**queue\_t** \*\*) &taskQueue, (**queue\_t**\*) task);  #**if** defined(DEBUG) || defined(DEBUGlite) // Mensagem de debug  UARTprintf("task\_create: tarefa %d (%s) - criada com sucesso**\n**", task->id, task->name);  #endif  **return** task->id;  }  // Alterna a execução para a tarefa indicada  **int** **task\_switch** (**task\_t** \*task)  {  **task\_t** \*taskAux = taskAtual;  taskAtual = task;  #ifdef DEBUG  UARTprintf("task\_switch: mudanca de contexto em andamento: %d->%d**\n**", taskAux->id, task->id);  #endif  swap\_context\_asm(&(taskAux->context),&(task->context)); // Mudança do contexto em execucao  **return** **0**;  }  // Retorna o identificador da tarefa corrente (main deve ser 0)  **int** **task\_id** ()  {  #ifdef DEBUG  UARTprintf("task\_id: tarefa atual: %d **\n**", taskAtual->id);  #endif  **return** taskAtual->id; // Retorna o id da tarefa em execucao  }  // Encerramento de tarefa  **void** **task\_exit** (**int** exitCode)  {  #**if** defined (DEBUG)|| defined(DEBUGlite)  UARTprintf("task\_exit: encerramento da tarefa %d (%s) **\n**", taskAtual->id, taskAtual->name);  #endif  **if** (taskAtual->id == dispatcher)  task\_switch(taskMain);  **else** {  **if** (taskQueue){  **task\_t** \*taskFinalizada = (**task\_t** \*) queue\_remove((**queue\_t** \*\*) &taskQueue, (**queue\_t** \*) taskAtual);  free (taskFinalizada);  }  task\_switch(&taskdisp);  }  }  // Aciona o despachante  **void** **task\_yield** ()  {  #ifdef DEBUG  UARTprintf("task\_yeld: chamada pela tarefa %d (%s)**\n**", taskAtual->id, taskAtual->name);  #endif  **if** (taskAtual->id != main && taskAtual->id != dispatcher){  queue\_append((**queue\_t** \*\*) &taskQueue, (**queue\_t** \*) taskAtual);  }  task\_switch (&taskdisp);  } |

Figura 2: Código “ppos\_core.c”

A figura 03 abaixo apresenta a execução do programa na porta serial da placa TIVA.

Gráfico de dispersão

Descrição gerada automaticamente

Figura 3: execução P3

O código é baseado na atividade *PingPongOS* do Prof. Carlos Maziero, usado para ensinar aos alunos o funcionamento rudimentar de um sistema operacional. Essencialmente o funcionamento gira em torno de uma estrutura de dados chamada *task*, que se comporta como um *Process Control Block*, a qual são feitas atribuições especificas á atividade em questão. Essas *structs* são manipuladas por *tasks* do sistema, sendo uma delas o escalonador, usando filas e trechos de código específicos para que o sistema mostre ao usuário na execução, uma simulação de um escalonamento de *threads* que imprimem palavras na tela, de forma transparente. Dentre os trechos do código, destaca-se a função “*task\_yield()*”, que por sua vez chama outra função “*task\_switch()*”. A anterior é a função que os processos chamam para ceder o seu controle ao despachante, e a posterior é a *task* chamada para realizar a troca de contexto entre os processos. Além disso temos também a função “*task\_create()*” que é responsável por criar as *tasks* em si, atribuindo à *struct* os valores de parâmetros definidos pelo usuário nos argumentos da chamada.

A figura 3 mostra que a função *task\_yield()*; acontece de forma “programada” por assim dizer, visto que está no espaço de *thread body*, é uma função que toda *thread* executa quando é chamada pelo escalonador. O motivo de esse ponto ser levantado é de que é perceptível que o sistema em questão não é preemptivo. Isso porque o sistema não é capaz de voluntariamente tomar o controle de uma *task* para ceder a outra. De certa forma, pode-se dizer que o escalonamento é “*hard-coded*” no sistema.

**3. P5 – Preempção por tempo**

A prática P5 envolve adicionar um sistema de preempção por tempo no código do *OS* para que, ao invés de fazer com que as *tasks* precisem chamar *task\_yield()* para continuar com o escalonamento, temos que um *timer* realiza uma “contagem regressiva” toda vez que uma *task* é escalonada, servindo como um “*quantum*” de tempo, e logo após o esgotamento do *timer*, a função é preemptada.

Infelizmente não conseguimos fazer este código funcionar corretamente na placa TIVA com êxito. A Solução para este problema foi adaptar os arquivos e códigos, e utilizar implementações anteriores de exemplo de forma que fosse possível compilar em um sistema *Linux*, como era feito anteriormente nas práticas de *software*.

O código permanece muito similar, com algumas alterações em lugares específicos. No código do despachante, foi implementada uma função que simula processamento pesado. A figura 4 abaixo ilustra essa mudança.

|  |
| --- |
| **void** **Body** (**void** \* arg)  {  **int** i,j ;  **for** (i=**0**; i<**10**; i++)  {  printf ("%s %d**\n**", (**char** \*) arg, i) ;  **for** (j=**0**; j<**10000000**; j++) ;  }  printf ("%s FIM**\n**", (**char** \*) arg) ;  task\_exit (**0**) ;  } |

Figura 4: mudança no código do dispatcher

Analisando a mudança percebemos que agora as *tasks* não cedem mais o controle utilizando *task\_yield()*, e realizam um laço *for* de uma contagem com um valor alto. Agora devem ser feitas alterações para que o sistema conte quanto tempo o processo fica ativo.

A figura 5 abaixo demonstra o exerpto do código “*pingpong.c*”, alterado para se adequar a esta tarefa.

|  |
| --- |
| **void** **dispatcher\_body** (**void** \*arg) // dispatcher é uma tarefa  {  **int** size\_of\_queue = (**int**)queue\_size((**queue\_t**\*)queue\_ready);  // Enquanto houverem tarefas prontas para serem executadas o loop continua  **while** ( size\_of\_queue > **0** )  {  **task\_t** \*task\_next;  task\_next = NULL;  // a funçao scheduler decide qual a procima tarefa a ser executada  task\_next = scheduler() ; // scheduler retorna a proxima tarefa  **if** (task\_next != NULL){  // Remoção da proxima tarefa a ser executa da fila de prontos, por meio do casting para queue\_t  queue\_remove((**queue\_t**\*\*)&queue\_ready, (**queue\_t**\*)task\_next);  #ifdef DEBUG  printf("dispatcher\_body: Task %d foi removida da fila de prontos**\n**", task\_next->t\_id);  #endif  task\_next->ptr\_queue = NULL;  //O quantum é resetado para o valor padrão do sistema:  quantum = QUANTUM;  // a tarafa next é lançada  task\_switch (task\_next) ; // transfere controle para a tarefa "next"  // ... // ações após retornar da tarefa "next", se houverem  }  size\_of\_queue = (**int**)queue\_size((**queue\_t**\*)queue\_ready);  }  task\_exit(**0**) ; // encerra a tarefa dispatcher  }  **void** **signal\_handler**(**int** singnum){  //Se a tarefa atual for uma tarefa de usário ela poderá ser “preemptada” caso necessário  // então a cada chama do tratador seu quantum diminui  **if**(task\_current->t\_type == USER\_TASK){  **if**(quantum < **1**){  #ifdef DEBUG  printf("signal\_handler: Tarefa chegou ao final do quantum: %d**\n**", task\_corrente->tid);  #endif  task\_yield();  }  **else**{  quantum--;  }  }  **return**;  } |

Figura 5: mudanças do código OS

A vantagem de utilizar o *Linux* para essa aplicação é que podemos utilizar as bibliotecas “*signal.h*” e “*sys/time.h*” para realizar o tratamento de sinal da mesma forma que exemplificado pelo Prof. Maziero. A implementação consiste em um valor de *quantum* que é estabelecido nas definições do código. Esse *quantum* representa o tempo em que um processo pode permanecer ativo e escalonado. Usando as exemplificações do professor Maziero, é utilizado um temporizador configurado como abaixo.

|  |
| --- |
| //Inicialização de timer e sinais e demais tratamentos para os mesmos  #ifdef DEBUG  printf("Setando timers e sinais**\n**")  #endif  action.sa\_handler = signal\_handler ;  sigemptyset (&action.sa\_mask) ;  action.sa\_flags = **0** ;  **if** (sigaction (SIGALRM, &action, **0**) < **0**)  {  printf ("Erro em pingpong\_init(): Erro em sigaction**\n** ") ;  exit (**1**) ;  }  // ajusta valores do temporizador  timer.it\_value.tv\_usec = **100** ; // primeiro disparo, em micro-segundos  timer.it\_value.tv\_sec = **0** ; // primeiro disparo, em segundos  timer.it\_interval.tv\_usec = **1000** ; // disparos subsequentes, em micro-segundos  timer.it\_interval.tv\_sec = **0** ; // disparos subsequentes, em segundos  // arma o temporizador ITIMER\_REAL (vide man setitimer)  **if** (setitimer (ITIMER\_REAL, &timer, **0**) < **0**)  {  printf ("Erro em pingpong\_init(): Erro em setitimer **\n**") ;  exit (**1**) ;  } |

Figura 6: temporizador

Este temporizador irá “contar” quanto tempo a *task* fica escalonada. Cada milissegundo é decrementado o valor de *quantum*. Quando o valor de *quantum* se esgota, é feito um *task\_yield()*, ou seja, quando a *task* esgota o *time slice*, é trocada. O resultado dessa implementação se dá na imagem abaixo.

Uma imagem contendo Texto

Descrição gerada automaticamente

Figura 7: execução com preempção

Analisando em um primeiro instante, parece que apenas foram organizadas para serem executadas na mesma sequência, porém mais vezes. Na verdade, no entanto, o que acontece é que a *task* está realizando seu *loop* várias vezes porque ela AINDA ESTÁ alocada. Percebemos que a *task Pang* termina em 2, depois quando é reescalonada termina em 8, e depois, numa parte que foi obscurecida no print, termina as últimas instâncias. Isso ocorre porque, como agora a *task* não faz *yield* manualmente, ela continua fazendo seu trabalho até ser preemptada. O *loop* *for* apresentado na figura 4, onde a *task* conta de 0 até 10000000, indiretamente afeta isso, pois quanto maior o número que deve ser contado, mais tempo leva para a *task* realizar uma iteração, o que também depende do *clock* do processador. Vale lembrar que este laço *for* apenas simula uma tarefa de alto custo computacional, em situações de maior gasto, e situações mais realistas, a quantidade de iterações das *tasks* antes de serem preemptadas poderiam ser menores, necessitando mais tempo total para executar o código.

**4. Conclusão: A diferença prática entre P3 e P5**

Como afirmado anteriormente, a prática P3 trata de um sistema não preemptivo, onde as *tasks* cedem controle apenas quando suas funções a fazem ceder. Por mais que na prática aparente ser um sistema de escalonamento “organizado”, não se trata de real preempção. Na P5 estamos falando de verdadeira preempção, pois ao invés de as *tasks* cederem o controle quando o *thread body* definir, o despachante em si toma o controle da task e cede para outra. Essa diferenciação de quem retém o real controle é o que torna o sistema de P5 preemptivo, e o sistema de P3 não preemptivo

**5. Referências**

- MAZIERO, C. A. **Preempção e compartilhamento de tempo.** Disponível em:<https://wiki.inf.ufpr.br/maziero/doku.php?id=so:preempcao_por_tempo>. Acesso em: julho de 2024.

- MAZIERO, C. A. **Despachante de tarefas.** Disponível em:<https://wiki.inf.ufpr.br/maziero/doku.php?id=so:dispatcher>. Acesso em: julho de 2024.