Esta ficha foi elaborada para acompanhar o estudo da disciplina de Sistemas de Operação (a4s1) ou para complementar a preparação para os momentos de avaliação finais da mesma. Num segundo ficheiro poderás encontrar um conjunto de propostas de solução aos exercícios que estão nesta ficha. É conveniente relembrar que algum conteúdo destes documentos pode conter erros, aos quais se pede que sejam notificados pelas vias indicadas na página web, e que serão prontamente corrigidos, com indicações de novas versões.

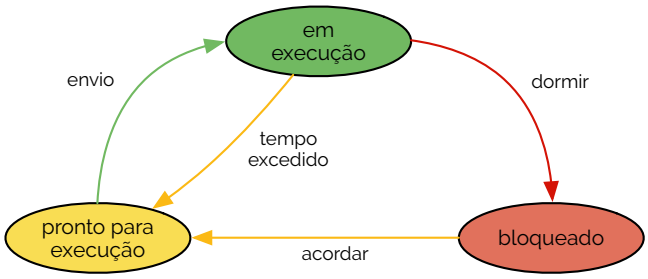
1. Distingue multiprocessamento e multiprogramação.

Quando referimos multiprocessamento devemos ser capazes de processar várias tarefas em simultâneo, e não concorrer para um mesmo recurso, numa ideia de paralelismo. Por outro lado, com a multiprogramação existe a capacidade de multiplexar no tempo a atribuição do(s) processador(es) para os diferentes programas presentes, numa ideia de concorrência.

2. Havendo o conceito de multiprogramação, em termos de sequência de instruções a executar, tendo em conta três programas A, B e C qual é o papel do sistema operativo ao tentar executar os três programas em simultâneo?

Sendo que se aplica a multiprogramação, então cabe ao sistema operativo saber como comutar entre as várias execuções dos vários programas A, B e C, inserindo, a nível de instruções, passos para comutação de execução para os programas A, B ou C. Por exemplo, considerando que uma parte do programa A está a ser executado e que, num dado instante, A não usará o processador, em termos de sequência de instrução o sistema operativo irá introduzir um conjunto de instruções para a execução do programa B ou C, enquanto o A não necessita do processador.

3. O escalonador de processador de baixo-nível típico possui 3 estados, normalmente designados de "em execução", "pronto para execução" e "bloqueado". Traça o diagrama de estados para um escalonador de baixo-nível, considerando os estados anteriores e, para cada transição considerada, explica o seu papel e quando é que ocorre.



No diagrama acima, quando um processo se encontra pronto a ser executado, em fila de espera, e se for submetido a envio (dispatch), então será selecionado para execução. Já na fase de execução, uma de duas alternativas poderão acontecer: se se despoletar um sinal de tempo excedido (timer-run-out), então o processo volta para a fila de espera no estado pronto-a-executar; por outro lado, se houver um sinal de dormir (sleep), então este terá de transitar para o estado bloqueado. Neste último estado, o processo, para avançar, só terá de receber um sinal de acordar (wake up), de forma a poder transitar para o pronto-a-executar.

4. Qual é a importância de haver um escalonamento de nível médio, onde se gere uma área de swapping?

A área de swapping e, por conseguinte, a gestão de nível médio do processador, é importante porque a memória principal, por muito grande que seja, é necessariamente finita, o que é uma possível limitação em termos do número de processos a guardar nesta, dentro de um dos três estados definidos no exercício anterior. Assim, cria-se uma extensão à memória na memória de massa principal (disco rígido ou SSD).

5. Considera o seguinte programa "howManyProcesses.c".

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

int main() {
  printf("Hello World!\n");
  fork();
  printf("Hello World!\n");
  fork();
  printf("Hello World!\n");
  fork();
  printf("Hello World!\n");
  return 0;
}
```

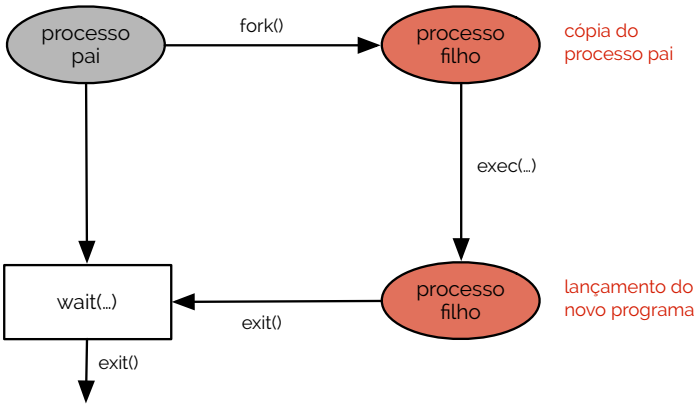
a) Quantos processos, considerando o processo inicial, é que são criados no programa acima?

Considerando o processo inicial são criados 8 processos. Ora o processo pai conta como um processo. Quando a execução passa pelo primeiro fork() é criado um processo-filho, totalizando um número de 2 processos. Quando a execução passa pelo segundo fork() é criado um processo por cada um dos processos criados anteriormente, pelo que ficam 4 processos. Depois, ao passar no terceiro fork() é criado um processo por cada um dos processos anteriormente criados, pelo que ficam 8 processos.

b) Quantas vezes é imprimida a mensagem "Hello World!"?

A mensagem "Hello World!" é imprimida 15 vezes. Inicialmente é imprimido pelo processo-pai (1 vez). Depois do primeiro fork, cada um dos processos imprime a mensagem - como existem dois processos então a mensagem é imprimida por ambos (3 vezes). Ao passar no segundo fork(), com 4 processos, cada um imprime (3 + 4 = 7 vezes). No fim, com 8 processos, cada um imprime 1 vez, pelo que totalizamos 7 + 8 = 15 vezes.

c) Traça o diagrama de criação e morte de um processo e usa-o para explicar o que acontece no programa.



Inicialmente temos um processo-pai que cria um processo-filho, através do fork(), copiando-se integralmente e identificando-se de forma diferente (com diferente PID). Depois de executar o seu programa (várias invocações do exec(...), este termina com exit(), fundindo-se com o término do processo-pai, que espera pela morte dos filhos com a função wait(...), antes de terminar (exit()).

d) Se o código da função main fosse a seguinte, onde delay() é uma função que gera um atraso com tempo aleatório em busy waiting, considerando que a execução do programa daria um output de "0 1 2 3 4" e que os printf() nunca bloqueam o processo, o processo-pai pode passar pelo estado "bloqueado"?

```
int main() {
  printf("0 ");
  int pid = fork();
  switch (pid) {
    case 0:
      delay();
      printf("1 ");
      printf("2 ");
      break;
    default:
      delay();
      printf("3 ");
      wait(NULL);
      printf("4 ");
  }
}
```

Sim, o pai pode passar pelo estado "bloqueado". Inicialmente o programa irá imprimir "0 ". Continuando, ao passar no fork(), uma cópia do processo-pai é realizada e uma seguirá com um pid de 0 (processo-filho) e outro com pid igual a um número, para nós arbitrário. Assim sendo, enquanto que o processo-filho, que dado o atraso (delay) inicia os prints primeiro que o processo-pai, imprime "1 " e "2 ", seguido do pai que imprime o "3 " e fica bloqueado à espera que o filho termine, podendo este ainda ter que executar o break e terminar a função. Só depois é que o pai poderá imprimir "4 ", ficando, no fim "0 1 2 3 4 ".

6. Dizemos que os processos em UNIX estão num estado "zombie" se já terminaram, mas se o processo-pai ainda não invocou wait(). Qual é o interesse de guardar os processos como zombie na tabela de controlo de processos (PCT)?

A chamada de sistema wait() permite a um processo-pai a espera pelo término de um dos seus filhos e conhecer o seu valor de retorno (definido pela chamada exit() ou por um return() da função main do fiilho), tal como o seu PID. Se um dos seus filhos já tiver morrido, então o wait() retorna de imediato.

Conservar os processos zombies na PCT permite que sejam conservados os seus PIDs e os seus valores de retorno, que possam ser necessários para o processo-pai. Este armazenamento é um pouco caro, dado que todos os outros recursos de sistema associados aos processos já foram libertados pelo sistema no momento da sua terminação. Esta é, no entanto, uma forma bastante simples para que o processo-pai não crie nenhum outro processo com um mesmo PID que já tivera sido lançado antes como seu filho, não criando objeto para ambiguidades aquando da invocação do wait() e das informações de retorno por PID dos seus filhos.

7. Quando um processo invoca um fork() qual dos seguintes conceitos é que é partilhado entre o processo-pai e o processo-filho: stack, heap ou segmentos partilhados de memória? O que é que acontece aos outros dois conceitos?

Claramente, e tal como o nome o indica, somente os segmentos partilhados de memória é que são partilhados entre o processo-pai e seus filhos. Os outros conceitos são criados como cópias dos originais (tanto a stack como a heap são recriados por cada filho).

8. Qual é a importância das políticas de escalonamento?

Quando referimos políticas de escalonamento pretendemos identificar uma razão para a escolha do envio (ou dispatch) dos processos que aguardam execução no estado "pronto para execução". As políticas de escalonamento são assim importantes para garantirem questões de justiça, previsibilidade, throughput, tempo de resposta, tempo de turnaround, deadlines e eficiência.

9. Existem várias políticas de escalonamento com definição de prioridades quer estáticas como dinâmicas.

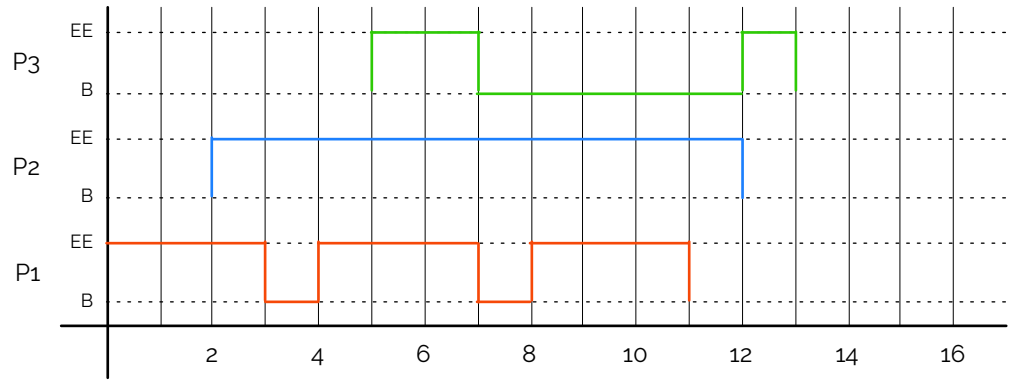
a) Distingue políticas de escalonamento com prioridade estática de políticas de escalonamento com prioridade dinâmica.

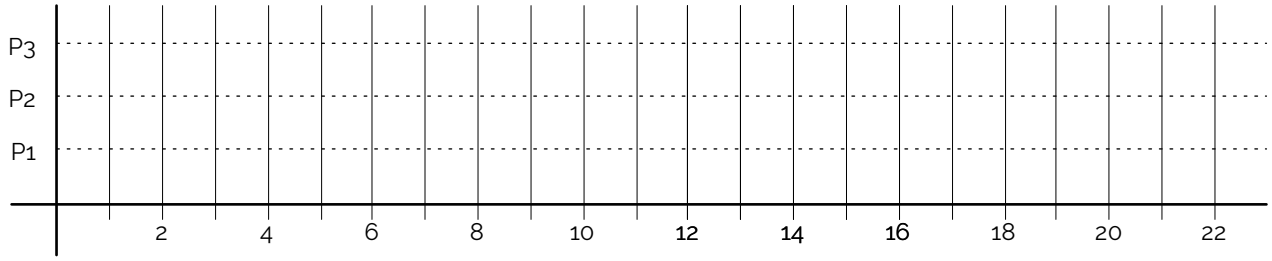
Na definição de prioridades existem duas possíveis alternativas: prioridades estáticas, onde o seu método de definição é puramente determinístico; prioridades dinâmicas, onde os vários métodos de definição possuem histerese, isto é, dependem da história da execução de um determinado processo.

b) Compara as políticas de escalonamento do processador designadas de FCFS e de round-robin.

Ambas as políticas de escalonamento FCFS (sigla de First-Come, First-Served) e round-robin são tais que se ajustam ao critério da justiça entre processos. Basicamente, o que ambas fazem é colocar os processos em pé de igualdade e jogar, depois, com a possibilidade da antecipação. O que se pretende referir com a antecipação é que, sem ela, os processos quando chegam são logo antendidos (numa política FCFS). Por outro lado, com antecipação, como é o caso da política round-robin, ao processos são atribuídas janelas temporais máximas para utilização dos recursos do processador.

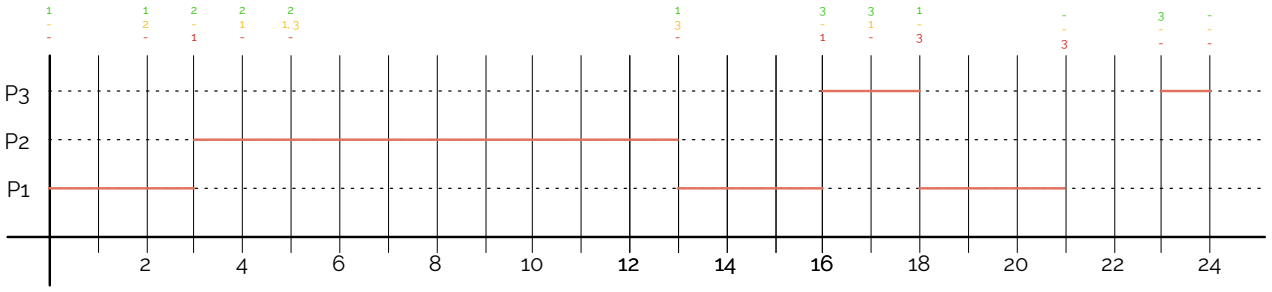
c) Considera o estado de execução de 3 processos independentes entre si (mesmo em termos de I/O), P1, P2 e P3, assumindo que correm em processadores (virtuais) distintos. EE e B indicam, respetivamente, quando é que os processos se encontram "em execução" e "bloqueados". Considera agora, para o gráfico seguinte, que os 3 processos estão a ser executados num ambiente monoprocessador e traça o diagrama temporal de escalonamento dos processos P1, P2 e P3 com uma política FCFS.





Para começarmos este exercício é importante termos em mente o diagrama do escalonador de baixo-nível, com os estados "em execução", "pronto a executar" e "bloqueado". Nestas soluções vamos apresentar uma solução com os estados dos processos representados por texto a verde, amarelo e vermelho, respetivamente, de tempos a tempos.

Inicialmente, com uma política FCFS, temos que o processo P1 necessita do processador, colocando o processo P1 em execução. Em t=2 o processo P2 necessita do processador, mas como o P1 já está em execução, este terá de aguardar na fila dos "pronto a executar", até que o P1 largue o processador. Isto acontece em t=3, onde P1 passa para o estado "bloqueado" (dado que tem uma unidade de tempo sem necessidade de utilização do processador) e P2 passa para execução. Como P2 não tem qualquer momento de pausa ao longo da sua execução, já podemos esperar que este ocupe o processador desde t=3 até t=13. Em t=4 o processador recebe o pedido de execução por parte do processo P1, que fica em "pronto para execução". Em t=5 o processo P3 pede acesso ao processador, o qual ficará retido em espera, a seguir ao P1, em "pronto para execução". Estando a fila de "pronto para execução" com os itens "P1, P3", por esta ordem, podemos saltar para o fim da execução de P2, em t=13, onde o primeiro item da lista de espera é atendido (P1) e executado. Em t=16 o processo P1 para a sua execução, passando para o estado "bloqueado" e deixando o processador livre para o processo P3 que estava à espera na fila "pronto a executar". Em t=17 o processo P1 passa para a fila de "pronto a executar" e em t=18 passa para execução, dado que P3 liberta o processador. Em t=21 P1 termina a sua execução, ficando apenas P3 no estado "bloqueado", deixando o processador inerte durante as próximas duas unidades de tempo (idle). Em t=23 o P3 passa para "em execução" e este termina em t=24, deixando o processador livre para outros processos.

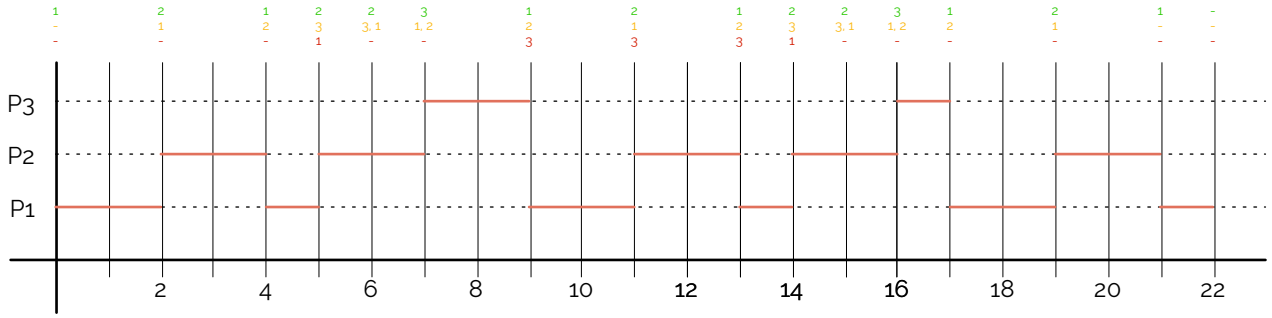


d) Com as mesmas considerações que na alínea anterior, traça o diagrama temporal de escalonamento dos processos P1, P2 e P3 com uma política de round-robin, com uma janela temporal máxima de 2 unidades de tempo.

Para começarmos este exercício é importante termos em mente o diagrama do escalonador de baixo-nível, com os estados "em execução", "pronto a executar" e "bloqueado". Nestas soluções vamos apresentar uma solução com os estados dos processos representados por texto a verde, amarelo e vermelho, respetivamente, de tempos a tempos.

Inicialmente, com uma política round-robin, temos que o processo P1 necessita do processador, colocando o processo P1 em execução. Como a janela temporal máxima é de duas unidades de tempo, então, em t=2 o processo P1 terá de sair de execução. Dada uma possível ambiguidade, neste ponto, e porque o processo P2 entra agora, a fila de "pronto para execução" poderá ter os processos "P2, P1" ou "P1, P2". No nosso caso, embora o caso contrário também possa ser possível, vamos considerar que a fila estaria "P2, P1", pelo que em t=2 entra em execução o processo P2 e P1 fica na fila de espera. Esta entrada em execução é feita por mais 2 unidades de tempo, sendo que em t=4 ficamos com o processo P1 a ser executado e com P2 a reentrar na fila de espera, pela transição "tempo excedido". Em t=5 o processo P1 bloqueia, o processo P2 entra em execução e o processo P3 pede para ser executado, entrando na fila de espera. Como o processo P1 apenas fica bloqueado por 1 unidade de tempo, em t=6 volta para a fila de espera, sendo que em t=7 a janela temporal do processo P2 é esgotada e o processo P3 toma conta do processador por duas unidades de tempo, deixando a fila de espera com "P1, P2". Passadas 2 unidades de tempo, isto é, em t=9 o processo P1 passa para execução, P2 permanece em espera e P3 passa para execução. (...)

Em suma, o gráfico abaixo mostra a comutatividade entre os vários processos num ambiente monoprocessador.



e) Distingue processos CPU-intensivos de I/O-intensivos e classifica P1, P2 e P3 dentro dessas categorias.

Processos CPU-intensivos são tais que ocupam grande parte do seu tempo de execução a usar o CPU (têm o CPU como recurso principal de processamento). Por outro lado, processos I/O-intensivos ocupam grande parte do seu tempo em interações com módulos de I/O. No diagrama dos processos P1, P2 e P3, o processo P2 é claramente CPU-intensivo e o P3 é I/O-intensivo. Dado que o tempo de CPU é semelhante ao tempo restante, no caso do processo P1 não é possível indicar se é CPU-intensivo ou I/O-intensivo (ambas as respostas estariam corretas).

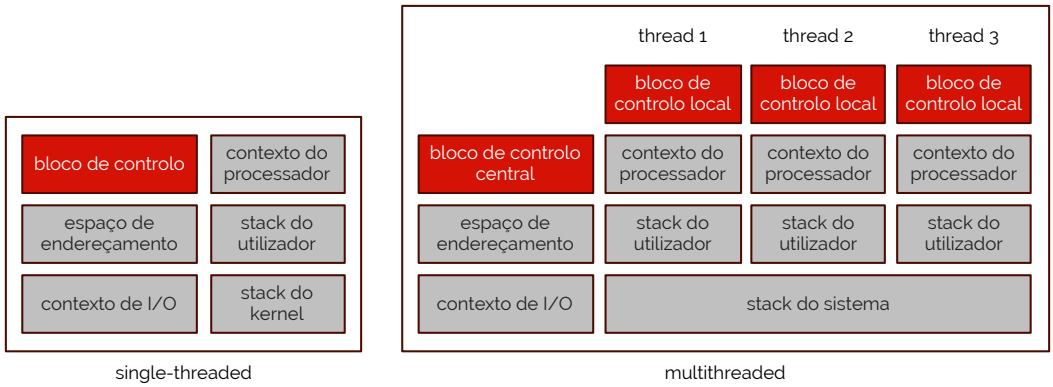
10. Qual é o significado do termo busy-waiting? Que outros tipos de espera existem num sistema de operação? O busy-wait pode ser evitado totalmente?

O busy-waiting significa que um processo está à espera que uma condição seja satisfeita num ciclo estrito sem renunciar ao processador. Alternativamente, um processo poderá esperar renunciando ao processador e bloqueando numa determinada condição, à espera que alguém o acorde num outro tempo no futuro. O busy-waiting pode (e deve) ser evitado totalmente, mas depois obtemos um overhead causado pelas operações de sinalizar processos que se encontram em espera.

11. Qual é a diferença entre processo e thread? Qual das seguintes duas afirmações é que está correta: "Um processo pode estar relacionado com um ou mais threads" ou "Uma thread está relacionada com um ou mais processos"?

Um processo é uma instância de posse de um recurso para a execução de um programa. Quando nos referimos a um processo estamos a identificar uma ou mais marcas de execução de um programa através, por exemplo, de um identificador único (PID). Uma thread, por outro lado, é um fio de execução de um programa, podendo haver mais do que um por processo. Assim sendo, a afirmação correta é "um processo pode estar relacionado com um ou mais threads".

12. Considera a figura abaixo.



a) O que é que pretende representar a figura acima?

A figura acima pretende representar as diferenças entre os ambientes single-threaded e multithreaded. Num sistema single-threaded, existe um bloco de controlo global, um determinado espaço de endereçamento (do processo), tal como um contexto de I/O, uma stack de sistema (ou do kernel) e uma stack própria para a utilização da thread pelo utilizador e um contexto próprio para o processador. Já num ambiente multithreaded podemos ver que a stack do sistema passa a ser partilhada pelas várias threads, sendo que por cada uma destas surge uma stack de utilizador, um contexto de processador e um bloco de controlo único por thread. Isto permite que em termos de identificação (posses) haja um processo que seja responsável (através do espaço de endereçamento) e vários fios de

execução sejam tomados como independentes (têm stacks de utilizador, contextos de processador e blocos de controlo local diferentes).

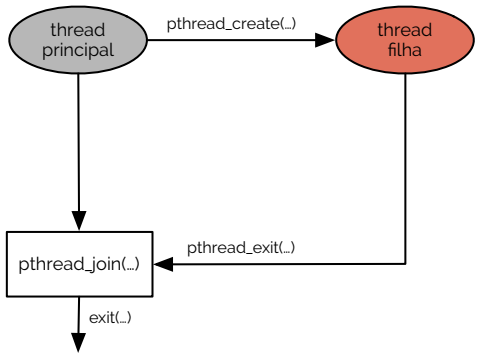
b) A biblioteca pthread, do UNIX, permite a manipulação de threads no espaço de utilizador. Considera o código abaixo:

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

static pthread_t threadA, threadB, threadC;
static int status = 0;

void* printBye(void* arg) {
  printf("bye! ");
  status = EXIT_SUCCESS;
  pthread_exit(&status);
}

void* printOtherMessages(void* arg) {
  printf("2 ");
  pthread_create(&threadC, NULL, printBye, NULL);
  status = EXIT_SUCCESS;
  pthread_exit(&status);
}

void* printMessages(void* arg) {
  printf("1 ");
  status = EXIT_SUCCESS;
  pthread_exit(&status);
  pthread_create(&threadA, NULL, printOtherMessages, NULL);
}

int main() {
  printf("begin ");
  pthread_create(&threadB, NULL, printMessages, NULL);
  int* status_p;
  pthread_join(threadA, (void**) &status_p);
  pthread_join(threadB, (void**) &status_p);
  pthread_join(threadC, (void**) &status_p);
  printf("end ");
  return EXIT_SUCCESS;
}
```

i) Qual é a primeira thread a ser executada: thread A, thread B ou thread C?

A primeira thread a ser executada é a thread B, na segunda linha da função main().

ii) Traça o diagrama de uma thread concordando com a biblioteca pthread.



iii) Qual é o output esperado deste programa?

No início da execução é imprimido "begin " e é criada a thread B, regida pela função printMessages(). Esta função começa por imprimir "1 " e termina logo a sua execução, através da invocação do exit(). O que acontece é que isto fecha a execução da thread, condicionando a execução do pthread_create(threadA). Feito o join(), é imprimido "end ", ficando no fim um output "begin 1 end ".