## Programa vs. Processo

Um programa é essencialmente um conjunto de instruções que descrevem como uma tarefa deve ser realizada por um computador. No entanto, por si só, um programa não executa a tarefa. É como um plano de voo que precisa ser posto em prática.

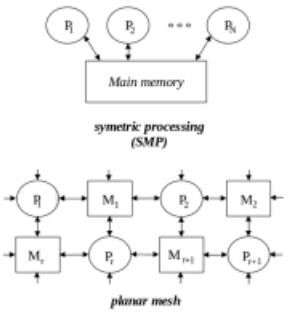
O processo é a entidade que representa a execução real de um programa. Ele encarna uma atividade específica e é caracterizado por vários elementos:

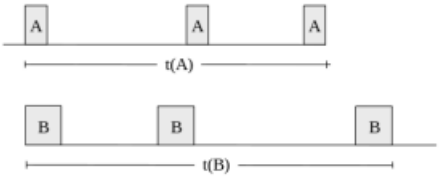
- Espaço de Endereçamento: Contém o código e os dados do programa, incluindo os valores reais das variáveis.
- Dados de Entrada e Saída: Incluem informações que fluem dos dispositivos de entrada para os de saída, desempenhando um papel fundamental na interação do processo com o ambiente.
- Variáveis Específicas do Processo: Identificadores únicos, como PID (Identificador do Processo) e PPID (Identificador do Processo Pai), que ajudam no gerenciamento e controle dos processos.
- Valores Atuais dos Registos Internos do Processador: Reflete o estado atual da execução, fornecendo uma visão instantânea do progresso do processo.

## Multiprocessamento

O paralelismo representa a habilidade de um sistema computacional executar simultaneamente dois ou mais programas. Essa proeza exige mais de um processador, com cada um dedicado a uma execução simultânea.

Para viabilizar o multiprocessamento, é imperativo ter mais de um processador disponível. Cada processador é destinado a uma execução específica, permitindo que programas distintos operem em paralelo. Os sistemas operativos desses ambientes são projetados para suportar eficazmente o multiprocessamento.



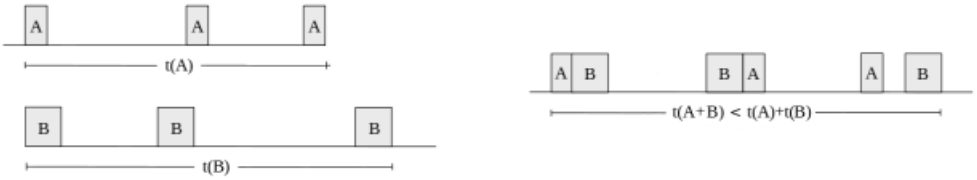




Imagine um sistema com pelo menos dois processadores. Nesse ambiente, os programas A e B podem ser executados em paralelo. Enquanto o programa A avança em um processador, o programa B realiza suas operações simultaneamente em outro processador.

## Multiprogramação

A concorrência representa a ilusão habilmente criada por um sistema computacional, dando a impressão de poder executar simultaneamente mais programas do que o número real de processadores disponíveis. Essa façanha é alcançada ao designar os processadores existentes para diferentes programas de maneira multiplexada no tempo.

Para tornar possível a multiprogramação, é necessário um sistema operativo que suporte eficazmente essa abordagem. Nele, os processadores existentes são alocados de maneira temporal a diferentes programas. Enquanto um programa está em execução, outros podem aguardar na fila, prontos para ocupar o processador quando for a sua vez.




Imagine um sistema com apenas um processador. Neste ambiente, os programas A e B podem executar aparentemente simultaneamente, pois o processador alterna entre eles em intervalos de tempo. Enquanto o programa A está em execução, o programa B aguarda pacientemente pela sua vez, criando a sensação de concorrência.

## Processos no Unix

Vamos explorar um exemplo simples:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

int main(void) {
   printf("Hello, World!\n");
   fork();
   printf("Hello, World! Again\n");
   return EXIT_SUCCESS;
}
```

No código acima, a função fork() desempenha um papel crucial. Ela clona o processo em execução, criando uma réplica idêntica. Ambos os processos compartilham inicialmente o mesmo espaço de endereçamento.

Logo após o fork(), os dois processos têm espaços de endereçamento iguais. Inicialmente, são idênticos. Geralmente, é seguido um método de cópia sob demanda (copy-on-write), onde as alterações no espaço de endereçamento são realizadas apenas quando necessário, economizando recursos.

Os estados de execução, incluindo o valor do contador de programa, são os mesmos após o fork(). Os processos continuam a execução a partir do mesmo ponto. A linha seguinte ao fork() será executada por ambos.

## Processos no Unix

Vamos explorar um exemplo simples:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

int main(void) {
   printf("Before the fork:\n");
   printf("PID = %d, PPID = %d.\n", getpid(), getppid());

   fork();

   printf("After the fork:\n");
   printf("PID = %d, PPID = %d.\nAm I the parent or the child? How can I know
it?\n", getpid(), getppid());

   return EXIT_SUCCESS;
}
```

Antes do fork(): O processo original imprime o PID (Identificador do Processo) e PPID (Identificador do Processo Pai).

Após o fork(): Ambos os processos (pai e filho) imprimirão o PID e PPID. No entanto, o desafio é determinar qual é o pai e qual é o filho.

Após o fork(), algumas variáveis de processo diferem entre o pai e o filho. O PID e PPID são exemplos claros. Essas diferenças permitem distinguir entre o processo pai e o processo filho.

Mas o que podemos fazer com isto? Com esta capacidade de criar processos filhos, podemos explorar cenários como execução paralela, comunicação entre processos e a construção de aplicações mais complexas que envolvem múltiplos fluxos de execução.

## Processos no Unix

Vamos explorar um exemplo simples:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

int main(void) {
   printf("Before the fork:\n");
   printf("PID = %d, PPID = %d.\n", getpid(), getppid());

   int ret = fork();

   if (ret == 0) {
      printf("I'm the child:\n");
      printf("PID = %d, PPID = %d\n", getpid(), getppid());

      // Executar um programa diferente no filho usando exec
      execl("/bin/ls", "ls", NULL);
   } else {
      printf("I'm the parent:\n");
      printf("PID = %d, PPID = %d\n", getpid(), getppid());

      // Esperar pelo término do processo filho
      wait(NULL);
   }

   return EXIT_SUCCESS;
}
```

Fork e Identificação: O fork() é usado para criar um novo processo. O valor retornado é usado para distinguir entre o processo pai e o processo filho.

Exec System Call: No processo filho, utilizamos a chamada de sistema execl para executar um programa diferente. Neste caso, estamos executando o comando ls.

Wait System Call: No processo pai, utilizamos a chamada de sistema wait para esperar pelo término do processo filho antes de continuar.

O fork por si só pode parecer de pouco interesse, mas quando combinado com exec e wait, abre portas para a execução de programas diferentes em processos filhos e a coordenação eficaz entre processos pai e filho.

Como no código anterior, assumimos que o fork não falha. Em caso de erro, ele retorna -1.

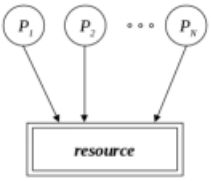
## Key concepts: Independent and Interacting Processes

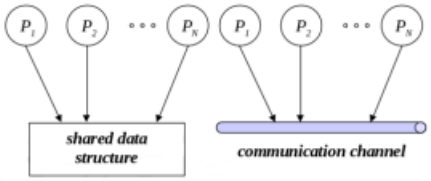
Independent Processes:

Em ambientes multiprogramados, dois ou mais processos podem ser considerados independentes se, desde a sua criação até a terminação, nunca interagirem explicitamente. No entanto, é importante notar que há uma interação implícita, pois esses processos competem por recursos do sistema. Um exemplo disso ocorre em sistemas de lotes, onde diferentes trabalhos são executados sem interação direta.

Interacting Processes:

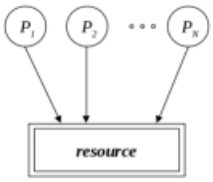
Em contraste, processos interativos envolvem a partilha de informações ou comunicação explícita. Isso geralmente requer um espaço de endereçamento comum, onde os processos podem trocar dados. A comunicação entre esses processos pode ocorrer através desse espaço de endereçamento compartilhado ou por meio de canais de comunicação dedicados.



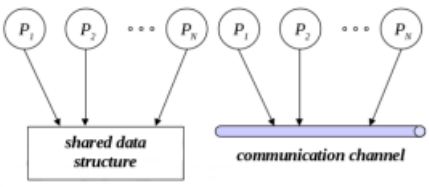

## Key Concepts: Independent and Interacting Processes (2)

Independent Processes Competing for a Resource:




- Em um ambiente multiprogramado, processos independentes competem por recursos.
- A responsabilidade do sistema operacional (SO) é garantir que a atribuição de recursos aos processos seja feita de maneira controlada, evitando perda de informações.
- Geralmente, isso implica que apenas um processo pode usar o recurso de cada vez, garantindo acesso mutuamente exclusivo.
- O canal de comunicação é tipicamente um recurso do sistema, o que significa que os processos competem por seu uso.

Interacting Processes Sharing Information or Communicating:



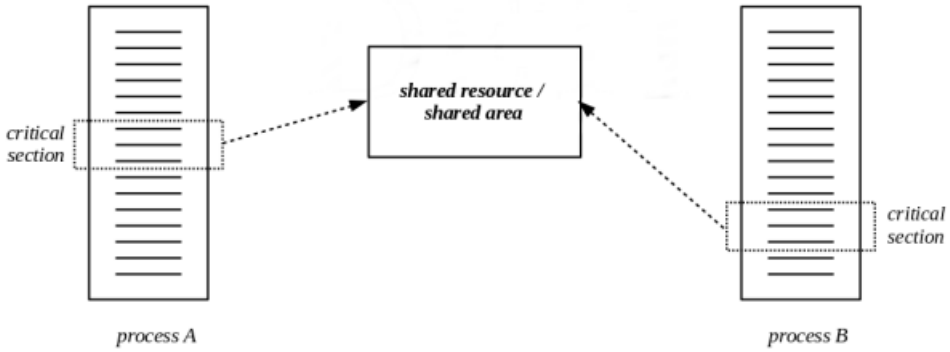

- Em contraste, processos interativos compartilham informações ou comunicam entre si.
- A responsabilidade recai sobre os processos para garantir que o acesso à área compartilhada seja feito de maneira controlada, evitando perda de informações.
- Normalmente, isso impõe que apenas um processo pode acessar a área compartilhada por vez, garantindo acesso mutuamente exclusivo.
- O canal de comunicação, muitas vezes, é um recurso do sistema, resultando em competição entre os processos por seu uso.

## Secção Crítica

Uma secção crítica refere-se a uma parte do código onde ocorre o acesso a um recurso ou área partilhada. Ter acesso a um recurso ou área partilhada implica executar o código específico dentro desta secção crítica.

É vital proteger adequadamente a secção crítica para evitar condições de corrida. Condições de corrida surgem quando o comportamento, saída ou resultado de um programa depende da sequência ou timing de eventos incontroláveis.

As consequências das condições de corrida podem resultar em comportamentos indesejáveis, tornando os resultados do programa imprevisíveis ou incorretos. Portanto, garantir a execução em exclusão mútua é fundamental. A exclusão mútua assegura que apenas um processo pode executar a secção crítica de cada vez, prevenindo conflitos e mantendo a integridade dos recursos partilhados.




## Conceitos-Chave: Deadlock e Starvation

Deadlock:

- A exclusão mútua no acesso a um recurso ou área partilhada pode resultar em deadlock.
- Um deadlock ocorre quando dois ou mais processos ficam permanentemente impedidos de aceder às suas respetivas secções críticas, aguardando por eventos que demonstradamente nunca ocorrerão.
- As operações ficam bloqueadas, e os processos envolvidos ficam numa espécie de impasse.

Starvation:

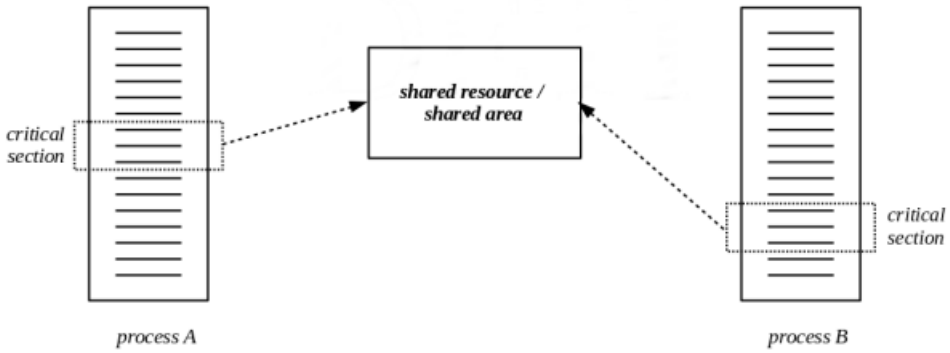
- Starvation acontece quando um ou mais processos competem pelo acesso a uma secção crítica, e, devido a uma conjunção de circunstâncias em que continuamente surgem novos processos que os ultrapassam, o acesso é sucessivamente adiado.
- As operações são continuamente adiadas, e os processos podem ficar à espera indefinidamente, mesmo que não haja deadlock.

## Memória Compartilhada como um Recurso

No âmbito dos sistemas operacionais, a memória compartilhada representa um recurso crucial, facilitando a comunicação e colaboração entre processos independentes. Embora os espaços de endereçamento dos processos permaneçam independentes, a natureza virtual desses espaços permite que a mesma região física seja mapeada em dois ou mais domínios virtuais.

Gerida pelo sistema operacional, a memória compartilhada envolve duas ações fundamentais. Primeiramente, os processos devem solicitar um segmento de memória compartilhada ao SO. Em seguida, o segmento alocado é mapeado no espaço de endereçamento do respectivo processo.

Esse mecanismo permite que os processos compartilhem informações de maneira fluida, utilizando o mesmo espaço de memória. Como resultado, tarefas colaborativas e comunicação eficiente entre processos tornam-se possíveis, aprimorando a funcionalidade geral e a coordenação dentro de um ambiente computacional.




## Unix IPC Primitives - Memória Compartilhada

System V Shared Memory:

- Criação: Utiliza a função shmget.
- Mapeamento e Desmapeamento: Utiliza as funções shmat e shmdt.
- Outras Operações: Envolvem a função shmctl.

POSIX Shared Memory:

- Criação: Envolvem as funções shm_open e ftruncate.
- Mapeamento e Desmapeamento: Utiliza as funções mmap e munmap.
- Outras Operações: Incluem close, shm_unlink, fchmod, e outras.

Estas primitivas fornecem mecanismos eficazes para a implementação de memória compartilhada no ambiente Unix, permitindo a comunicação e partilha de dados entre processos de maneira eficiente.

## Problema do Buffer Limitado

No âmbito deste problema, um conjunto de entidades (produtores) gera informações que são consumidas por diversas entidades distintas (consumidores).

A comunicação ocorre através de um buffer com capacidade limitada, partilhado por todas as entidades envolvidas. Supõe-se que cada produtor e cada consumidor operem como processos distintos.

Consequentemente, a implementação do FIFO (First-In-First-Out) deve ser realizada em memória partilhada para que os diferentes processos possam aceder a ele. Este FIFO representa a fila de itens produzidos pelos produtores e consumidos pelos consumidores.

## Semáforos: Mecanismo de Sincronização

Os semáforos emergem como uma ferramenta vital no arsenal da sincronização em sistemas operacionais e ambientes multitarefa. Definidos por um tipo de dados que incorpora duas operações atómicas, "down" e "up", os semáforos são projetados para coordenar o acesso concorrente a recursos partilhados.

O tipo de dados subjacente a um semáforo é estruturado da seguinte forma:

```
typedef struct {
   unsigned int val;
   PROCESS *queue;
} SEMAPHORE;
```

As operações fundamentais dos semáforos são as seguintes:

down:

- Bloqueia e enfileira o processo se o valor do semáforo for zero.
- Decrementa o valor do semáforo se este for diferente de zero.

up:

- Incrementa o valor do semáforo.
- Se a fila associada ao semáforo não estiver vazia, acorda um processo em espera, seguindo uma política predefinida.

Esta abstração poderosa e flexível permite a implementação eficaz de primitivas de sincronização, contribuindo para a prevenção de condições de corrida e garantindo a coordenação harmoniosa entre processos. Ao atuar como um guardião de recursos partilhados, os semáforos desempenham um papel vital na robustez e eficiência de sistemas operacionais modernos.

## Implementação Possível de Semáforos

Aqui está uma possível implementação de semáforos, utilizando um código em pseudocódigo para ilustrar:

```
typedef struct {
   unsigned int val;
   PROCESS *queue;
} SEMAPHORE;
```

Explicação:

- Nesta implementação, é assumido que R representa o número total de semáforos.
- A estrutura SEMAPHORE contém um valor (val) e uma fila de processos (queue).
- A função sem_down é responsável por diminuir o valor do semáforo. Se o valor for zero, bloqueia o processo atual na fila associada ao semáforo (block_on_sem).
- A função sem_up aumenta o valor do semáforo. Se a fila associada ao semáforo não estiver vazia, acorda um processo dessa fila (wake_up_one_on_sem).

Características da Implementação:

- Esta implementação é típica de sistemas uniprocessadores, onde a concorrência entre processos é gerida através de interrupções.
- Semáforos podem ser binários ou não binários, dependendo das necessidades de sincronização.
- Para implementar exclusão mútua usando semáforos, um semáforo binário pode ser utilizado. Isso impede que mais de um processo aceda a uma secção crítica simultaneamente.

Esta implementação fornece uma base para a gestão eficaz de recursos partilhados e sincronização entre processos em sistemas operacionais ou ambientes multitarefa.

## Unix IPC Primitives - Semaphores

System V Semaphores:

- Criação: Utiliza a função semget.
- Operações "down" e "up": São realizadas pela função semop.
- Outras Operações: Incluem semctl.

POSIX Semaphores:

Existem dois tipos: semáforos com nome e sem nome.

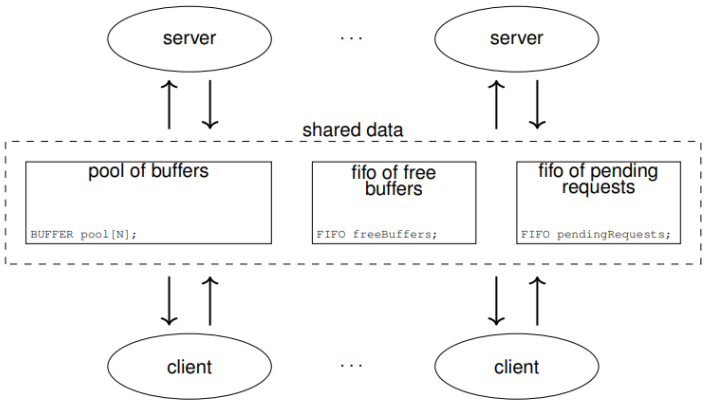
- Semáforos com Nome:
    - Operações incluem sem_open, sem_close, e sem_unlink.
    - São criados num sistema de ficheiros virtual (por exemplo, /dev/sem).
- Semáforos sem Nome:
    - Operações envolvem sem_init e sem_destroy.
    - "Down" e "up" são realizados por sem_wait, sem_trywait, sem_timedwait, e sem_post.

## Problema do Buffer Limitado - Implementação Segura Utilizando Semáforos

```
void fifoInsert(FIFO *f, uint32_t id, uint32_t v1, uint32_t v2)
{
   /* Aguardar até que o fifo não esteja cheio */
   psem_down(f->sem, SLOTS);

   /* Bloquear o acesso ao fifo */
   psem_down(f->sem, LOCKER);

   /* Inserção */
   f->data[f->in].id = id;
   f->data[f->in].v1 = v1;
   bwDelay(f->dummyDelay); // Para aumentar a probabilidade de ocorrência de condições de corrida
   f->data[f->in].v2 = v2;
   f->in = (f->in + 1) % f->size;
   f->cheio = (f->in == f->out);

   /* Libertar o acesso ao fifo */
   psem_up(f->sem, LOCKER);

   /* Notificar que há mais um item disponível */
   psem_up(f->sem, ITEMS);
}

void fifiRetrieve(FIFO *f, uint32_t *idp, uint32_t *v1p, uint32_t *v2p)
{
   /* Aguardar até que o fifo não esteja vazio */
   psem_down(f->sem, ITEMS);

   /* Bloquear o acesso ao fifo */
   psem_down(f->sem, LOCKER);

   /* Recuperação */
   *idp = f->data[f->out].id;
   *v1p = f->data[f->out].v1;
   bwDelay(f->dummyDelay); // Para aumentar a probabilidade de ocorrência de condições de corrida
   *v2p = f->data[f->out].v2;
   f->out = (f->out + 1) % f->size;
   f->cheio = false;

   /* Libertar o acesso ao fifo */
   psem_up(f->sem, LOCKER);

   /* Notificar que há mais um slot disponível */
   psem_up(f->sem, SLOTS);
}
```

Este código apresenta uma implementação segura do Problema do Buffer Limitado utilizando semáforos. A função fifoInsert insere um item no FIFO, e fifoRetrieve retira um item do FIFO. Semáforos são utilizados para garantir a exclusão mútua e sincronização, evitando condições de corrida e assegurando o correto funcionamento do buffer limitado.

## Exemplificação - Exemplo Cliente-Servidor




Enunciado do Problema:

- Neste cenário, várias entidades (clientes) interagem com outras entidades (servidores) para solicitar um serviço.
- A comunicação é realizada através de um pool de buffers partilhados por todos.
- Cada produtor e consumidor deve ser executado como um processo diferente.
- Portanto, a estrutura de dados deve ser implementada em memória partilhada para que os diferentes processos possam aceder a ela.

Este problema ilustra a dinâmica de comunicação entre clientes e servidores, onde solicitações de serviço são feitas e atendidas através de um conjunto compartilhado de buffers. Cada produtor (cliente) e consumidor (servidor) opera como processos independentes, e a implementação em memória partilhada é crucial para possibilitar o acesso concorrente à estrutura de dados central, garantindo uma troca eficiente de informações no ambiente cliente-servidor.

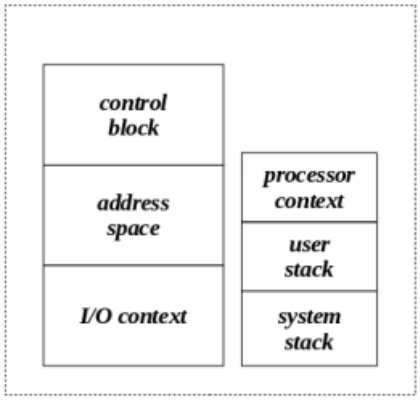
## Threads - Single Threading

Nos sistemas operativos tradicionais, um processo é uma entidade que compreende diversas partes fundamentais. Em primeiro lugar, existe um espaço de endereçamento que alberga o código e os dados do programa associado. Adicionalmente, o processo mantém um conjunto de canais de comunicação que facilitam a interação com dispositivos de entrada/saída. Contudo, o elemento distintivo é a presença de uma única thread de controlo. Esta thread incorpora os registos do processador, incluindo o contador de programa, e uma pilha que é utilizada para gerir as chamadas de função e as variáveis locais.

É crucial destacar que, apesar de estarem agrupados num só, estes componentes podem ser geridos independentemente uns dos outros. Neste modelo, uma thread surge como uma entidade de execução dentro de um processo, proporcionando uma abordagem mais flexível e granular.

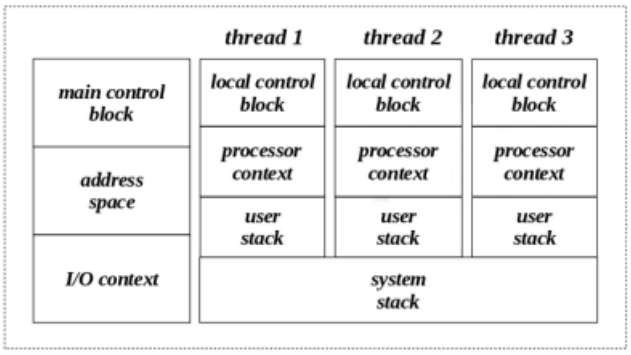
As threads possibilitam uma execução concorrente dentro de um processo, permitindo que diferentes partes do código sejam executadas em paralelo. Esta abordagem torna-se particularmente útil em cenários onde há tarefas que podem ser realizadas simultaneamente, otimizando assim o desempenho do sistema.

Em resumo, enquanto os processos representam a encapsulação global de recursos e funcionalidades, as threads oferecem uma unidade mais específica de execução, permitindo uma gestão mais eficiente e adaptável dos recursos do sistema.



*Single threading*

## Threads – Multithreading



*Multithreading*

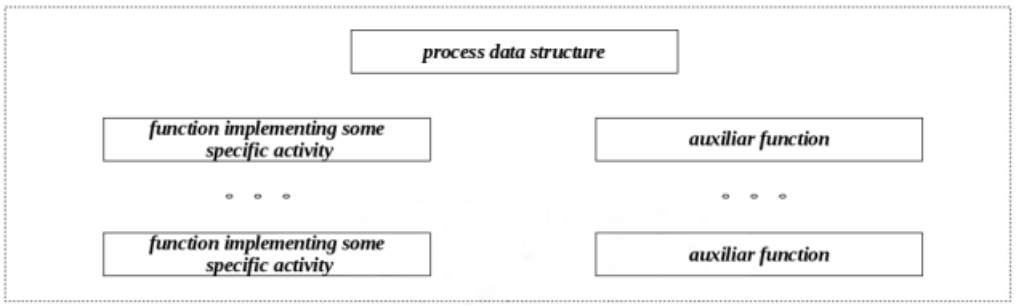
Multithreading Overview:

- Multithreading é a capacidade de ter várias threads independentes coexistindo no mesmo processo. Esta abordagem permite que as threads partilhem o mesmo espaço de endereçamento e o mesmo contexto de E/S (Entrada/Saída). Em vez de serem processos independentes, as threads dentro de um mesmo processo podem comunicar e cooperar mais eficientemente devido à partilha de recursos.

Lightweight Processes:

- As threads são frequentemente descritas como "lightweight processes" (processos leves), uma vez que consomem menos recursos do sistema do que processos completos. Cada thread tem o seu próprio conjunto de registos, mas partilha o mesmo espaço de endereçamento e recursos com outras threads no mesmo processo. Isto resulta numa execução mais eficiente e numa comunicação mais direta entre as threads.

## Threads - Structure of a Multithreaded Program




- Cada thread está tipicamente associada à execução de uma função que implementa uma atividade específica.
- A comunicação entre threads pode ser realizada através da estrutura de dados do processo, que é global do ponto de vista das threads.
- Inclui variáveis estáticas e dinâmicas (memória heap).
- O programa principal, também representado por uma função que implementa uma atividade específica, é a primeira thread a ser criada e, geralmente, a última a ser destruída.

## Threads - Implementations of Multithreading

User Level Threads:

- Neste modelo, as threads são implementadas por uma biblioteca ao nível do utilizador, proporcionando a criação e gestão de threads sem intervenção do kernel.
- Altamente versátil e portátil, este método oferece flexibilidade e independência do sistema operativo subjacente.
- Quando uma thread efetua uma chamada de sistema bloqueante, o processo inteiro fica bloqueado, uma vez que o kernel apenas perceciona o processo e não as threads individuais.

Kernel Level Threads:

- Neste modelo, as threads são implementadas diretamente ao nível do kernel.
- Menos versáteis e menos portáteis em comparação com as User Level Threads.
- Quando uma thread efetua uma chamada de sistema bloqueante, outra thread pode ser agendada para execução. Isto ocorre porque o kernel tem visibilidade direta sobre as threads individuais.

Ambas as abordagens possuem vantagens e desvantagens. As User Level Threads são mais flexíveis e portáteis, mas podem resultar em bloqueios completos do processo. Por outro lado, as Kernel Level Threads oferecem um agendamento mais granular, mas são menos versáteis e podem estar mais dependentes do sistema operativo subjacente.

## Threads - Advantages of Multithreading:

- Facilidades de Implementação de Aplicações: A implementação de aplicações torna-se mais simples ao decompor a solução em várias atividades paralelas. Esta abordagem facilita o modelo de programação, especialmente em aplicações complexas.
- Partilha Eficiente de Recursos: Devido à partilha do espaço de endereçamento e do contexto de E/S entre todas as threads, a multithreading favorece a decomposição de tarefas e a colaboração entre as threads. Isto simplifica a gestão de recursos ao permitir uma comunicação eficaz e uma partilha de dados.
- Gestão Aprimorada de Recursos Computacionais: A criação, destruição e troca de threads é mais simples do que realizar as mesmas operações com processos completos. A gestão mais eficiente de recursos é alcançada através da utilização de threads.
- Melhor Desempenho: Quando uma aplicação envolve considerável E/S, a multithreading possibilita a sobreposição de atividades, acelerando assim a execução. Esta sobreposição permite que as atividades prossigam enquanto outras estão à espera de operações de E/S, melhorando o desempenho global.
- Multiprocessamento: A multithreading viabiliza o real paralelismo em sistemas com múltiplas CPUs. A execução simultânea de várias threads é possível, tirando partido da capacidade de processamento paralelo.

## Threads no Linux – The clone system call:

No Linux, existem duas chamadas ao sistema para criar um processo filho:

- fork: Cria um novo processo que é uma cópia completa do processo atual. O espaço de endereçamento e o contexto de E/S são duplicados, e o processo filho inicia a execução no ponto do fork.
- clone: Cria um novo processo que pode partilhar elementos com o seu pai. O espaço de endereçamento, a tabela de descritores de ficheiros e a tabela de manipuladores de sinais são partilháveis. O processo filho inicia a execução numa função especificada.

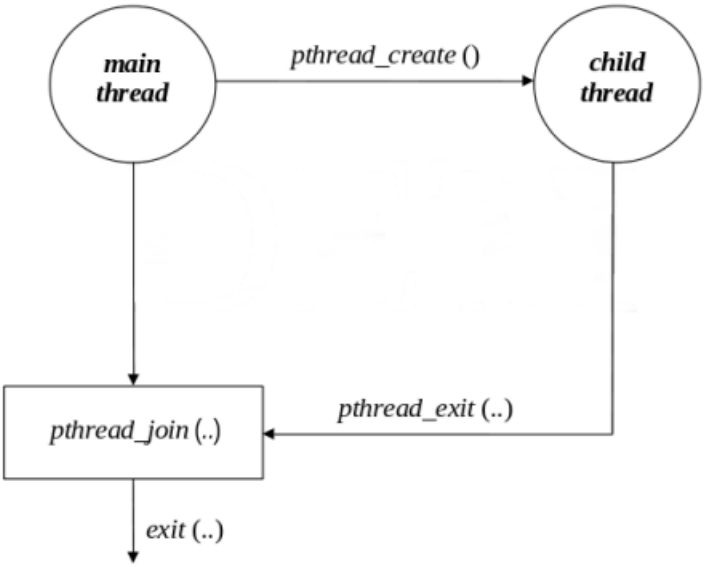
Deste modo, do ponto de vista do kernel, processos e threads são tratados de forma semelhante. Threads do mesmo processo formam um grupo de threads e têm o mesmo identificador de grupo de threads (TGID), que é o valor retornado pela chamada ao sistema getpid(). Dentro de um grupo, as threads podem ser distinguíveis pelo seu identificador de thread (TID) único, obtido através da chamada ao sistema gettid().

## Threads no Linux - POSIX

Funções Disponíveis para Gerir Threads:

- pthread_create: Cria uma nova thread, equivalente ao fork em processos.
- pthread_exit: Termina a thread que a chama, correspondendo ao exit em processos.
- pthread_join: Sincroniza com uma thread terminada, correspondendo ao waitpid em processos.
- pthread_kill: Envia um sinal a uma thread, equivalente ao kill em processos.
- pthread_cancel: Envias um pedido de cancelamento a uma thread.
- pthread_self: Obtém o ID da thread que a chama.

- pthread_detach: Desvincula uma thread, indicando que o sistema pode liberar recursos quando a thread terminar.




## Threads no Linux - Exemplo de Criação de Threads

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

/* Variável para armazenar o estado de término da thread */
int status;

/* Função para a thread filho */
void *threadChild(void *arg) {
   printf("I'm the child thread!\n");
   sleep(1);
   status = EXIT_SUCCESS;
   pthread_exit(&status);
}

/* Função principal (thread principal) */
int main(int argc, char *argv[]) {
   /* Lançamento da thread filho */
   pthread_t thread;
   if (pthread_create(&thread, NULL, threadChild, NULL) != 0) {
      perror("Fail launching thread");
      return EXIT_FAILURE;
   }
```

Este exemplo em C demonstra a criação e término de threads no ambiente Linux usando a biblioteca POSIX. A função threadChild é a função executada pela thread filho, enquanto a função main representa a thread principal. A thread filho imprime uma mensagem, aguarda um segundo e, em seguida, termina com um status específico.

O programa principal cria a thread filho, aguarda o seu término usando pthread_join e imprime o status retornado pela thread filho após o término. Este é um exemplo simples de como utilizar threads em Linux.

```
    /* Aguarda o término da thread filho */
    if (pthread_join(thread, NULL) != 0) {
        perror("Fail joining child");
        return EXIT_FAILURE;
    }

    printf("Child ends; status %d.\n", status);
    return EXIT_SUCCESS;
}
```

## Sincronização de Threads - Introdução a monitores

Problema com Semáforos:

- Um problema com semáforos é que eles são usados tanto para implementar exclusão mútua quanto para sincronização.
- Sendo primitivas de baixo nível, são aplicadas numa perspetiva bottom-up.
- Se as condições necessárias não forem satisfeitas, os processos são bloqueados antes de entrar em suas seções críticas.
- Esta abordagem é propensa a erros, principalmente em situações complexas, onde os pontos de sincronização podem estar dispersos pelo programa.

Abordagem de Monitor:

- Uma abordagem de nível superior deve seguir uma perspetiva top-down.
- Os processos devem primeiro entrar em suas seções críticas e depois aguardar se as condições de continuação não forem satisfeitas.
- Uma solução é introduzir uma construção (concorrente) ao nível de programação que lida com exclusão mútua e sincronização separadamente.
- Um monitor é tal mecanismo de sincronização, independentemente proposto por Hoare e Brinch Hansen, apoiado por uma linguagem de programação (concorrente).
- A biblioteca pthread fornece primitivas que permitem implementar monitores (do tipo Lampson-Redell).

## Sincronização de Threads - Definição de Monitor

```
typedef struct {
  /* Estrutura de dados interna partilhada */
  DATA data;

  /* Variável de condição */
  cond_t c;

  /* Métodos de acesso */
  void (*method_1)(...);
  void (*method_2)(...);

  /* Código de inicialização */
} monitor_example;
```

No contexto da sincronização de threads, um monitor é conceptualizado como um mecanismo que orquestra o acesso a uma estrutura de dados partilhada dentro de uma aplicação. Aqui estão os principais atributos e comportamentos associados a um monitor:

Perspetiva da Aplicação:

- Uma aplicação é percebida como uma coleção de threads competindo pelo acesso a uma estrutura de dados partilhada.
- A estrutura de dados partilhada só é acessível exclusivamente através de métodos de acesso designados.

Exclusão Mútua:

- Cada método é executado dentro de um domínio de exclusão mútua, garantindo que apenas uma thread de cada vez pode aceder à estrutura de dados partilhada.

Bloqueio de Execução:

- Se uma thread tenta chamar um método de acesso enquanto outra thread está atualmente dentro de um método de acesso diferente, a execução da primeira thread é interrompida até que a segunda thread complete o seu acesso.

Sincronização Através de Variáveis de Condição:

- A sincronização entre threads é alcançada através do uso de variáveis de condição.
- Duas operações principais em variáveis de condição são possíveis:
    - Espera: Esta operação resulta na bloqueação da thread e na sua remoção do monitor.
    - Sinal: Quando existem threads bloqueadas, emitir um sinal desperta uma delas. Surge a questão: qual delas deve ser despertada?

Esta abordagem estruturada para a sincronização de threads através de monitores garante acesso ordenado e consistente a dados partilhados, mitigando condições de corrida e promovendo uma cooperação eficaz entre as threads. A gestão cuidadosa da exclusão mútua e das variáveis de condição é fundamental para a implementação bem-sucedida de sistemas concorrentes.

## Sincronização de Threads – POSIX

A biblioteca pthread possibilita a implementação de monitores em C/C++. Para alcançar a exclusão mútua, são utilizados mutexes (elementos de exclusão mútua), enquanto variáveis de condição são empregues para a sincronização. Aqui estão algumas funções para suporte à exclusão mútua:

- pthread_mutex_t: O tipo de dados para o mutex.
- pthread_mutex_init: Inicializa um objeto mutex.
- pthread_mutex_lock: Bloqueia o mutex fornecido.
- pthread_mutex_unlock: Desbloqueia o mutex fornecido.

Para suporte à sincronização, algumas funções incluem:

- pthread_cond_t: O tipo de dados para a variável de condição.
- pthread_cond_init: Inicializa um objeto de variável de condição.
- pthread_cond_wait: Desbloqueia atomicamente o mutex associado e aguarda a sinalização da variável de condição fornecida.
- pthread_cond_signal: Reinicia uma das threads que estão à espera da variável de condição fornecida.
- pthread_cond_broadcast: Reinicia todas as threads que estão à espera da variável de condição fornecida.

Essas funcionalidades permitem a criação de monitores eficazes, garantindo a exclusão mútua e a sincronização adequada entre as threads em ambientes concorrentes. O uso combinado de mutexes e variáveis de condição proporciona uma abordagem robusta para a gestão da concorrência em programas multi-threaded.

## Exemplo Ilustrativo: Problema do Barbeiro Adormecido

Neste problema, atribuído a Dijkstra, um conjunto de entidades (clientes) interage com outra entidade (barbeiro) para solicitar um serviço (corte de cabelo). Eis a declaração do problema:

- Se não houver clientes, o barbeiro adormece na cadeira de barbeiro.
- Ao entrar, se o barbeiro estiver a dormir, um cliente deve acordá-lo.
- Caso contrário, se o barbeiro estiver a trabalhar, o cliente sai se todas as cadeiras estiverem ocupadas ou senta-se se houver uma disponível.
- Após terminar um corte de cabelo, o barbeiro verifica a sala de espera para ver se há clientes à espera, adormecendo se não houver nenhum.

Este cenário representa um desafio clássico de sincronização em sistemas concorrentes, onde a gestão eficiente do acesso ao barbeiro e à sala de espera é crucial para evitar condições de corrida e garantir um comportamento consistente do sistema.