

## Processes in Unix/Linux

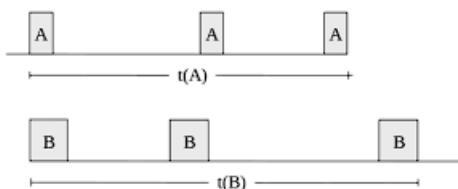
### - Programa vs. Processo

- **Programa** – conjunto de instruções que descrevem como uma tarefa é executada por um computador
  - Para que a tarefa seja realmente executada, o programa correspondente deve ser executado
- **Processo** – uma entidade que representa um programa de computador sendo executado
  - representa algum tipo de atividade
  - é caracterizado por:
    - **espaço de endereçamento** – código e dados (valores reais das diferentes variáveis) do programa associado
    - dados de entrada e saída (dados que estão sendo transferidos de dispositivos de entrada para dispositivos de saída)
    - processar variáveis específicas (PID, PPID, ...)
    - valores reais dos registos internos do processador
    - estado de execução
- Diferentes processos podem estar a executar o mesmo programa
- Em geral, existem mais processos do que processadores – **multiprogramação**

### - Multiprocessamento vs. Multiprogramação

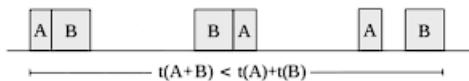
#### Multiprocessamento

- **Paralelismo** – capacidade de um sistema computacional executar simultaneamente dois ou mais programas
  - é necessário mais de um processador (um para cada execução simultânea)
- Os sistemas operacionais de tais sistemas computacionais suportam **multiprocessamento**

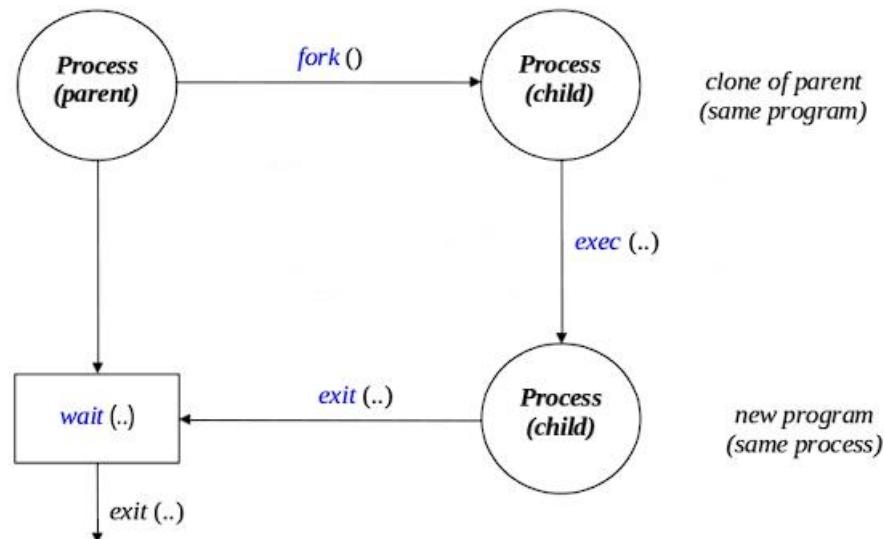


## Multiprogramação

- **Simultaneidade (Concurrency)** – ilusão criada por um sistema computacional de aparentemente ser capaz executar simultaneamente mais programas do que o número de processadores existentes
  - O(s) processador(es) existente(s) deve(m) ser atribuído(s) aos diferentes programas de uma vez de uma maneira multiplexada
  - Os sistemas operacionais de tais sistemas computacionais suportam **multiprogramação**



### - Creation by cloning



### - Process creation: fork0 & fork1

```
#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;
}

#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.\n"
           " Am I the parent or the child?"
           " How can I know it?\n",
           getpid(), getppid());

    return EXIT_SUCCESS;
}
```

- O fork clona a execução processo, criando uma réplica dele

- Os espaços de endereço dos dois processos são iguais
  - na verdade, logo após a bifurcação, eles são os mesmos
  - normalmente, uma abordagem **cópia por gravação (copy on write)** é seguida
- Os estados de execução são os mesmos
  - incluindo o valor do programa contador (programa counter)
- Algumas variáveis do processo são diferentes (PID, PPID, ...)

### - Process creation: fork2 and fork3

```

#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();

    printf("After the fork:\n");
    printf(" PID = %d, PPID = %d.\n",
           getpid(), getppid());
    printf(" ret = %d\n", ret);

    return EXIT_SUCCESS;
}

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());
    }
    else
    {
        printf("I'm the parent:\n");
        printf(" PID = %d, PPID = %d\n",
               getpid(), getppid());
    }

    return EXIT_SUCCESS;
}

```

- O valor retornado pelo fork é diferente em processos pai e filho
  - no pai, é o PID da criança
  - na criança, é sempre 0
- Este valor de retorno pode ser usado como uma variável booleana
  - para que possamos distinguir o código correndo na criança e nos pais

```

#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());
    }
    else
    {
        printf("I'm the parent:\n");
        printf(" PID = %d, PPID = %d\n",
               getpid(), getppid());
    }

    return EXIT_SUCCESS;
}

```

- Em geral, usado sozinho, o fork é de pouco interesse
- Em geral, queremos executar um programa diferente na criança
  - exec system call
  - existem diferentes versões de exec
- Às vezes, queremos que os pais aguardem a conclusão do programa em execução na criança
  - wait system call
- Neste código, estamos assumindo que o fork não falha
  - em caso de erro, retorna -1

### - Executing a C/C++ program: atexit

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

/* cleaning functions */
static void atexit_1(void)
{
    printf("atexit 1\n");
}

static void atexit_2(void)
{
    printf("atexit 2\n");
}

/* main program */
int main(void)
{
    /* registering at exit functions */
    assert(atexit(atexit_1) == 0);
    assert(atexit(atexit_2) == 0);

    /* normal work */
    printf("hello world 1!\n");

    for (int i = 0; i < 5; i++) sleep(1);

    return EXIT_SUCCESS;
}

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

int main(int argc, char *argv[], char *env[])
{
    /* printing command line arguments */
    printf("Command line arguments:\n");
    for (int i = 0; argv[i] != NULL; i++)
    {
        printf(" %s\n", argv[i]);
    }

    /* printing all environment variables */
    printf("\nEnvironment variables:\n");
    for (int i = 0; env[i] != NULL; i++)
    {
        printf(" %s\n", env[i]);
    }

    /* printing a specific environment variable */
    printf("\nEnvironment variable:\n");
    printf(" env[\"HOME\"] = \"%s\"\n", getenv("HOME"));
    printf(" env[\"zzz\"] = \"%s\"\n", getenv("zzz"));

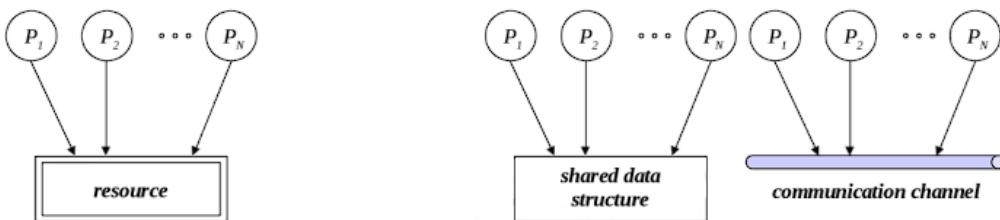
    return EXIT_SUCCESS;
}
```

- A função atexit permite registar uma função a ser chamada no programa terminação normal
- Eles são chamados na ordem inversa em relação ao seu registo
- argv é um array de strings
- argv[0] é o programa referência
- env é um array de strings,
- cada um representando uma variável, na forma par nome-valor
- getenv retorna o valor de um nome de uma variável

## Semaphores and shared memory

### - Key concepts : Independent and interacting processes

- Num ambiente multiprogramado, dois ou mais processos podem ser:
  - **independentes** – se eles, desde a sua criação até a sua extinção, nunca explicitamente interagiram
    - na verdade, há uma interação implícita, pois eles competem pelos recursos do sistema
    - ex: trabalhos em sistema batch; processos de diferentes usuários
  - **interativo** – se eles compartilham informações ou se comunicam explicitamente
    - o **compartilhamento** requer um **espaço de endereço comum**
    - a **comunicação** pode ser feita através de um espaço de endereço comum ou de um canal de comunicação conectando-os



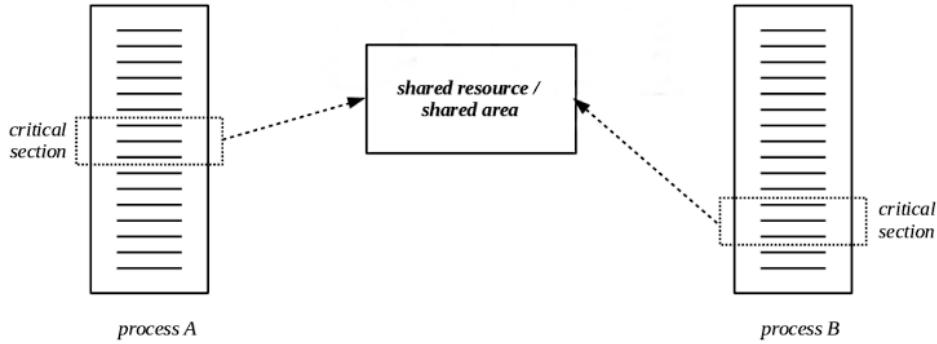
- **Processos independentes** competindo por um recurso
- É **responsabilidade do SO** garantir que a alocação de recursos para processos é feita de forma controlada, tal que nenhuma informação perdida ocorra
- Em geral, isso impõe que apenas um processo pode usar o recurso por vez – **acesso mútuo exclusivo**
- O canal de comunicação é normalmente um recurso do sistema, então os processos competem por isso

- **Processos interativos** compartilham informação ou comunicação
- É **responsabilidade dos processos** garantir que o acesso à área compartilhada seja feito de forma controlada, de modo que não se perda informações
- Em geral, isso impõe que apenas um processo pode acessar a área compartilhada em um tempo – **acesso mútuo exclusivo**
- O canal de comunicação normalmente é um recurso do sistema, então processos competem por isso

### - Critical Section

- Ter acesso a um recurso ou a uma área compartilhada significa, na verdade, **executar o código** que faz o acesso
- Esta seção do código, chamada **seção crítica**, se não for protegida adequadamente, pode resultar em **condições de corrida (race conditions)**

- Uma **condição de corrida** é uma condição em que o comportamento (saída, resultado) depende da sequência ou no timing de outros eventos (incontroláveis)
  - Pode resultar em comportamento indesejável
- **Seções críticas** devem ser executadas em **exclusão mútua**



#### - Deadlock and starvation

- A exclusão mútua no acesso a um recurso ou área compartilhada pode resultar em
  - **deadlock** - quando dois ou mais processos são permanentemente impedidos de acessar as suas respetivas seções críticas, esperar por eventos que possam ser demonstrados nunca irá acontecer
    - as operações estão bloqueadas
  - **starvation** - quando um ou mais processos competem pelo acesso a uma seção crítica e, devido a uma conjunção de circunstâncias em que surgem continuamente novos processos que os excedem, o acesso é sucessivamente bloqueado.
    - as operações são continuamente adiadas

#### - Shared memory

- Os espaços de endereçamento dos processos são independentes
- Mas os espaços de endereçamento são virtuais
- A mesma região física pode ser mapeada em duas ou mais regiões virtuais
- A **memória partilhada** é gerida como um recurso pelo sistema operativo
- São necessárias duas acções:
  - Solicitar um segmento de memória partilhada para o SO
  - Mapear esse segmento no espaço de endereço do processo

#### - Unix IPC primitives : Shared memory

- **System V shared memory**
  - creation – *shmget*
  - mapping and unmapping – *shmat*, *shmdt*
  - other operations – *shmctl*
  - execute *man shmget*, *man shmat* *man shmdt* or *man shmctl* for specific descriptions
  - execute *man 7 sysvipc* for an overview description

- POSIX shared memory

- creation – *shm\_open*, *ftruncate*
- mapping and unmapping - *mmap*, *munmap*
- other operations - *close*, *shm\_unlink*, *fchmod*, ···
- execute *man shm\_open*, *man mmap*, *man munmap*, *man shm\_close*,...  
for specific descriptions
- execute *man shm\_overview* for an overview description

### - Semaphores : Definition

- Um **semáforo** é um mecanismo de sincronização, definido por um tipo de dados mais duas operações atómicas, **down** e **up**

- Tipo de dados:

```
typedef struct
{
    unsigned int val;      /* can not be negative */
    PROCESS *queue;        /* queue of waiting blocked processes */
} SEMAPHORE;
```

- Operações

- Down
  - bloqueia e coloca o processo em fila de espera se *val* for zero
  - decremento de *val* caso contrário
- Up
  - incrementar *val*
  - se a fila não estiver vazia, desperta um processo em espera (de acordo com uma determinada política)

- Note-se que *val* só pode ser manipulado através destas operações
  - Não é possível verificar o valor de *val*

### - Semaphores : Analysis of semaphores

- As soluções concorrentes baseadas em semáforos têm vantagens e desvantagens
- **Vantagens:**
  - **suporte ao nível do sistema operativo** - as operações sobre semáforos são implementadas pelo kernel e disponibilizadas aos programadores como chamadas de sistema
  - **generalidade** - são construções de baixo nível e por isso são versáteis, podendo ser utilizados em qualquer tipo de solução
- Desvantagens:
  - **conhecimento especializado**- o programador deve ter conhecimento de princípios de programação concorrente, pois podem ser facilmente introduzidas condições de corrida ou deadlock

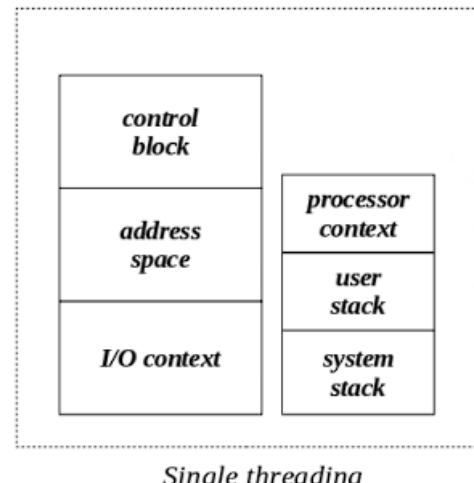
## - Unix IPC primitives : Semaphores

- System V semaphores
  - creation: *semget*
  - down and up: *semop*
  - other operations: *semctl*
  - execute *man semget*, *man semop* or *man semctl* for a description
- POSIX semaphores
  - Two types: named and unnamed semaphores
  - Named semaphores
    - *sem\_open*, *sem\_close*, *sem\_unlink*
    - created in a virtual filesystem (e.g., */dev/sem*)
  - unnamed semaphores – memory based
    - *sem\_init*, *sem\_destroy*
  - down and up
    - *sem\_wait*, *sem\_trywait*, *sem\_timedwait*, *sem\_post*
- execute *man sem\_overview* for an overview

## Threads, mutexes and condition variables in Unix/Linux

### - Threads Single threading

- No sistema operativo tradicional, um processo inclui:
  - um espaço de endereçamento (código e dados do programa associado)
  - um conjunto de canais de comunicação com dispositivos de I/O
  - uma single thread de controlo, que incorpora os registos do processador(incluindo o contador de programa) e uma pilha
- No entanto, estes componentes podem ser geridos separadamente
- Neste modelo, **thread** aparece como um componente de execução dentro de um processo

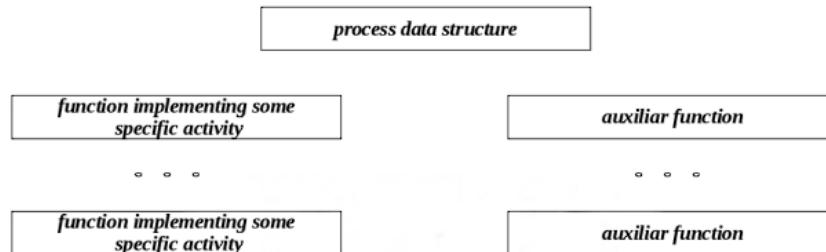


### - Threads : Multithreading

- Várias threads independentes podem coexistir no mesmo processo, partilhando assim o mesmo espaço de endereço e o mesmo contexto de I/O
  - Isto é designado por **multithreading**

- As threads podem ser vistos como **processos leves**

#### - Threads Structure of a multithreaded program



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

#### - Threads Implementations of multithreading

- **User level threads** - as threads são implementadas por uma biblioteca, ao nível do utilizador, que permite a criação e gestão de threads sem intervenção do kernel
  - versáteis e portáteis
  - quando uma thread chama uma chamada de sistema bloqueante (blocking system call), todo o processo bloqueia
    - porque o kernel só vê o processo
- **threads a nível do kernel** - as threads são implementadas diretamente a nível do kernel
  - menos versáteis e menos portáteis
  - quando uma thread chama uma chamada de sistema bloqueante, outra thread pode ser programada para execução

#### - Threads Advantages of multithreading

- **implementação mais fácil das aplicações** - em muitas aplicações, decompor a solução numa série de actividades paralelas torna o modelo de programação mais simples
  - uma vez que o espaço de endereçamento e o contexto de I/O são partilhados por todas as threads, o multithreading favorece esta decomposição.
- **melhor gestão dos recursos informáticos** - criar, destruir e mudar de thread é mais fácil do que fazer o mesmo com os processos
- **melhor desempenho** - quando uma aplicação envolve I/O substancial, o multithreading permite a sobreposição de actividades, acelerando assim a sua execução
- **multiprocessamento** - o paralelismo real é possível se existirem vários CPUs

#### - Threads in Linux : The *clone* system call

- No Linux existem duas chamadas de sistema para criar um processo filho:
  - `fork` - cria um novo processo que é uma cópia completa do atual
    - o espaço de endereço e o contexto de I/O são duplicados
    - o filho inicia a execução no ponto de forking
  - `clone` - cria um novo processo que pode partilhar elementos com o seu pai
    - o espaço de endereço, a tabela de descritores de ficheiros e a tabela de manipuladores de sinais são partilháveis.
    - o filho inicia a execução numa função especificada
- Assim, do ponto de vista do kernel, os processos e as threads são tratados da mesma forma
- As threads de um mesmo processo formam um grupo de threads e têm o mesmo identificador de grupo de threads (Tgid)
  - este é o valor devolvido pela chamada de sistema `getpid()`
- Dentro de um grupo, as threads podem ser distinguidas pelo seu identificador único de thread (tid)
  - este valor é devolvido pela chamada de sistema `gettid()`

#### - Thread in Linux : POSIX library

- Standard POSIX, IEEE 1003.1c, defines a programming interface (API) for the creation and synchronization of threads
  - In Linux, this interface is implemented by the `pthread` library
- Some of the available functions to manage threads:
  - `pthread_create` – create a new thread (corresponding to the `fork` in processes)
  - `pthread_exit` – terminate calling thread (corresponding to the `exit` in processes)
  - `pthread_join` – joint with a terminated thread (corresponding to the `waitpid` in processes)
  - `pthread_kill` – send a signal to a thread (corresponding to the `kill` in processes)
  - `pthread_cancel` – send a cancellation request to a thread
  - `pthread_self` – obtain ID of the calling thread
  - `pthread_detach` – detach a thread

#### - Thread synchronization : Introducing monitors

- Um problema com os semáforos é que eles são usados tanto para implementar a exclusão mútua e para sincronizar
- Sendo primitivos de baixo nível, são aplicados numa perspetiva ascendente (`bottom-up`)
  - se as condições exigidas não forem satisfeitas, os processos são bloqueados antes de entrarem nas suas secções críticas
  - esta abordagem é propensa a erros, principalmente em situações complexas, uma vez que os pontos de sincronização podem estar espalhados por todo o programa
- Uma abordagem de nível superior deve seguir uma perspetiva descendente (`top-down`)
  - os processos devem primeiro entrar nas suas secções críticas e depois esperar se as condições de continuação não forem satisfeitas

- Uma solução é introduzir uma construção (concorrente) ao nível da programação que trata da exclusão mútua e da sincronização separadamente
- Um **monitor** é um mecanismo de sincronização deste tipo, proposto independentemente por Hoare e Brinch Hansen, suportado por uma linguagem de programação (concorrente)
- A biblioteca pthread fornece primitivas que permitem implementar monitores (do tipo Lampson-Redell)

### - Thread synchronization Monitor definition

```
monitor example
{
    /* internal shared data structure */
    DATA data;

    cond c; /* condition variable */

    /* access methods */
    method_1 (...)

    {
        ...
    }

    method_2 (...)

    {
        ...
    }

    ...

    /* initialization code */
    ...
}
```

- An application is seen as a set of threads that compete to access the **shared data structure**
- This shared data can only be accessed through the access methods
- Every method is executed in **mutual exclusion**
- If a thread calls an access method while another thread is inside another access method, its execution is blocked until the other leaves
- Synchronization between threads is possible through **condition variables**
- Two operation on them are possible:
  - **wait** – the thread is blocked and put outside the monitor
  - **signal** – if there are threads blocked, one is waked up. *Which one?*

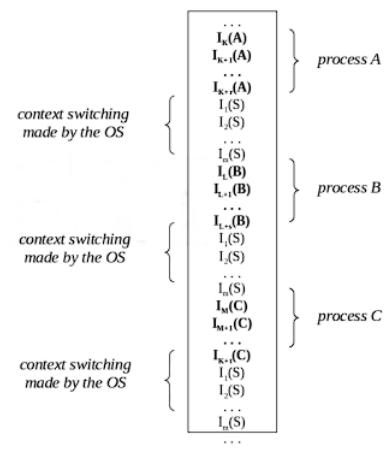
### - Thread synchronization POSIX support for monitors

- The **pthread** library allows for the implementation of monitors in C/C++
  - **mutexes** (mutual exclusion elements) are used to implement **mutual exclusion**
  - **condition variables** are used to implement **synchronization**
- Some function for mutual exclusion support:
  - **pthread\_mutex\_t** – the mutex data type
  - **pthread\_mutex\_init** – initializes a mutex object
  - **pthread\_mutex\_lock** – locks the given mutex
  - **pthread\_mutex\_unlock** – unlocks the given mutex
- Some function for synchronization support:
  - **pthread\_cond\_t** – the condition variable data type
  - **pthread\_cond\_init** – initializes a condition variable object
  - **pthread\_cond\_wait** – atomically unlocks the associated mutex and waits for the given condition variable to be signaled.
  - **pthread\_cond\_signal** – restarts one of the threads that are waiting on the given condition variable
  - **pthread\_cond\_broadcast** – restarts all of the threads that are waiting on the given condition variable

## Processes

### - Process model

- Na multiprogramação, a atividade do processador, porque está a mudar de um processo para outro, é difícil de perceber
- Assim, é melhor assumir a existência de um número de processadores virtuais, um por cada processo existente
  - número de processadores virtuais activos ≤ número de processadores reais
  - Desligar um processador virtual e ligar outro corresponde a uma comutação de contexto de comutação
- A comutação de contexto e, por conseguinte, a comutação entre processadores virtuais, pode ocorrer por diferentes razões, possivelmente não controladas pelo programa
- Assim, para ser viável, este modelo de processo exige que
  - a execução de qualquer processo não deve ser afetada pelo instante no tempo ou pela localização no código onde a comutação ocorre
  - não são impostas restrições aos tempos de execução total ou parcial de qualquer processo



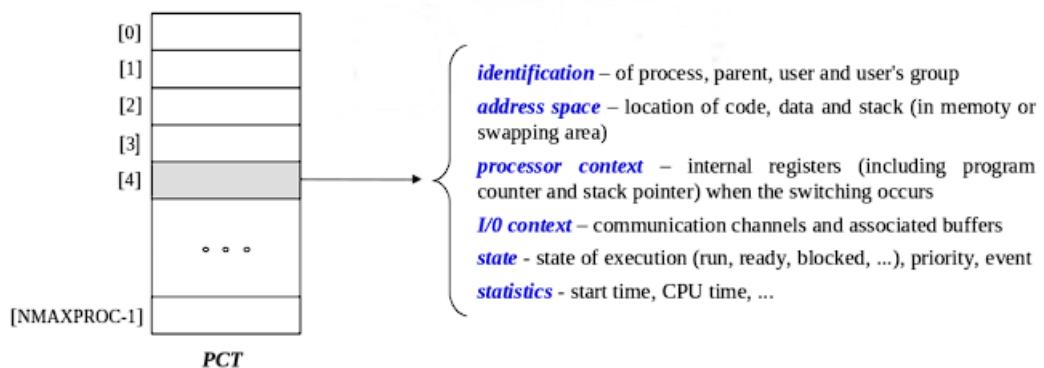
### - Context switching

- Os processadores actuais têm dois modos de funcionamento:
  - modo supervisor - todo o conjunto de instruções pode ser executado
    - é um modo privilegiado
  - modo utilizador - apenas uma parte do conjunto de instruções pode ser executada
    - as instruções de entrada/saída estão excluídas, bem como as que modificam os registos de controlo
    - é o modo normal de funcionamento
- A passagem do modo de utilizador para o modo de supervisor só é possível através de uma exceção (por razões de segurança)
- Uma exceção pode ser causada por
  - Interrupção de I/O
    - interrupção de I/O externa à execução da instrução atual
  - instrução ilegal (divisão por zero, erro de barramento)
    - associada à execução da instrução atual, mas não intencional
  - instrução trap (interrupção de software)
    - associada à execução da instrução atual, mas não prevista
- O sistema operativo deve funcionar em modo supervisor
  - para ter acesso a todas as funcionalidades do processador
- Assim, as funções do kernel (incluindo as chamadas de sistema) devem ser activadas por
  - hardware (interrupção)
  - armadilha (interrupção de software)

- Isto estabelece um ambiente de funcionamento uniforme: tratamento de exceções
- A mudança de contexto é o processo de armazenar o estado de um processo e restaurar o estado de outro processo
- A comutação de contexto ocorre necessariamente no contexto de uma exceção, com uma pequena diferença na forma como é tratada

#### - Process control table

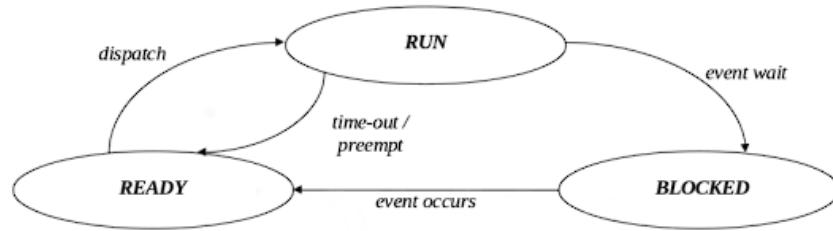
- Para implementar o modelo de processo, os sistemas operativos necessitam de uma estrutura de dados para armazenar as informações sobre cada processo – processo bloco de controlo
- A tabela de controlo de processos (TCP), que pode ser vista como uma matriz de blocos de controlo de processos, armazena informações sobre todos os processos



#### - (Short-term) Process states

- Um processo pode não estar a ser executado por diferentes razões
  - por isso, é necessário identificar os possíveis estados do processo
- Os mais importantes são:
  - **RUN** - o processo está na posse de um processador e, portanto, em execução
  - **BLOCKED** - o processo está a aguardar a ocorrência de um evento externo (acesso a um recurso, fim de uma operação de entrada/saída, etc.)
  - **READY** - o processo está pronto a ser executado, mas a aguardar a disponibilidade de um processador para iniciar/retomar a sua execução
- As transições entre estados resultam normalmente de uma intervenção externa, mas, em alguns casos, podem ser desencadeadas pelo próprio processo
- A parte do sistema operativo que trata destas transições é designada por **(processor) scheduler**, e é parte integrante do seu kernel
  - Existem diferentes políticas para controlar o disparo destas transições
  - Elas serão abordadas mais tarde

#### - Short-term state diagram

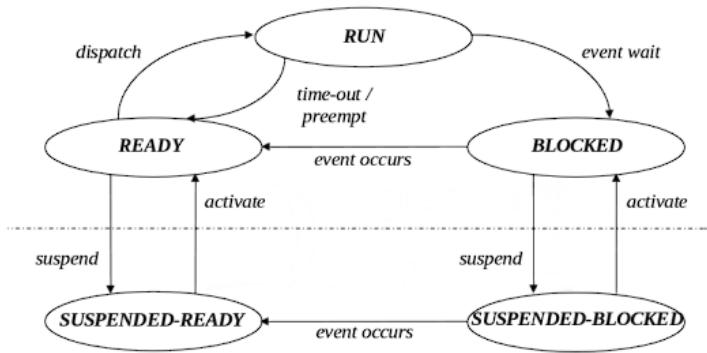


- espera de evento - o processo em execução é impedido de prosseguir, aguardando a ocorrência de um evento externo
- despacho - um dos processos prontos a correr é selecionado e é-lhe atribuído o processador
- ocorrência de evento - ocorreu um evento externo e o processo que o aguardava está agora pronto para receber o processador
- preempção - um processo de prioridade mais elevada prepara-se para ser executado, pelo que o processo em execução é removido do processador
- time-out - o quantum de tempo atribuído ao processo em execução chega ao fim, pelo que o processo é removido do processador

#### - Medium-term states

- A memória principal é finita, o que limita o número de processos coexistentes
- Uma forma de superar essa limitação é usar uma área na memória secundária para estender a memória principal
  - A esta área chama-se **área de troca** (pode ser uma partição do disco ou um ficheiro)
  - Um processo não em execução, ou parte dele, pode ser **trocado** para liberar a memória principal para outros processos
  - Esse processo será mais tarde **colocado em swap**, depois de a memória principal ficar disponível
- Dois novos estados devem ser adicionados ao diagrama de estados do processo para incorporar estas situações:
  - **suspensão-pronto** - o processo está pronto, mas foi trocado
  - **suspensão-bloqueado** - o processo está bloqueado e trocado

#### - State diagram, including short- and medium-term states

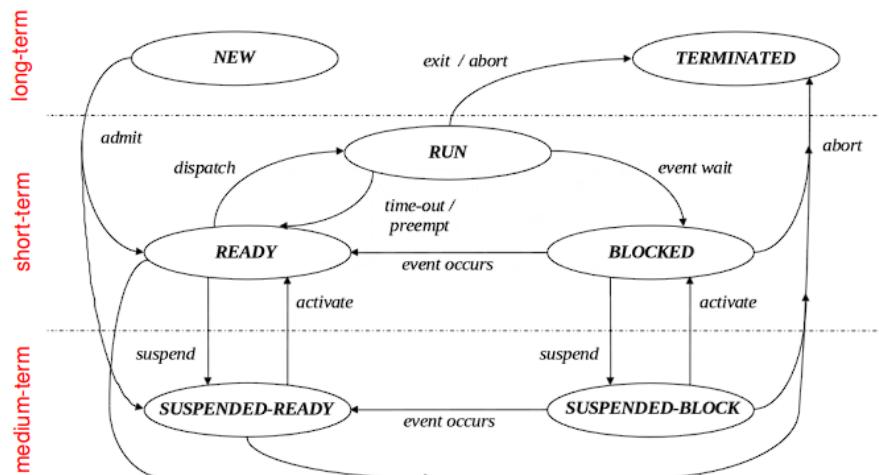


- Two new type of transitions appear:
  - **suspend** – the process is *swapped out*
  - **activate** – the process is *swapped in*

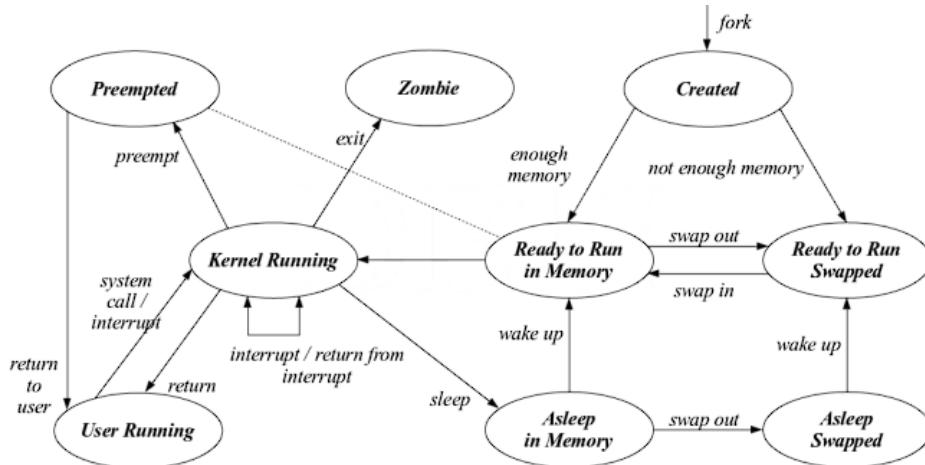
### - Long-term states and transitions

- O diagrama de estados anterior assume que os processos são intemporais
  - Com exceção de alguns processos do sistema, isto não é verdade
  - Os processos são criados, existem durante algum tempo e acabam por terminar
- Dois novos estados são necessários para representar a criação e o término
  - **new** - o processo foi criado, mas ainda não foi admitido no pool de processos executáveis (a estrutura de dados do processo foi inicializada)
  - **terminated** - o processo foi liberado do pool de processos executáveis mas ainda são necessárias algumas ações antes de o processo ser descartado
- existem três novas transições
  - **admit** - o processo é admitido (pelo OS) no conjunto de processos executáveis
  - **exit** - o processo em execução indica ao OS que foi concluído
  - **abort** - o processo é forçado a terminar (devido a um erro fatal ou porque um processo autorizado aborta a sua execução)

### - Global state diagram

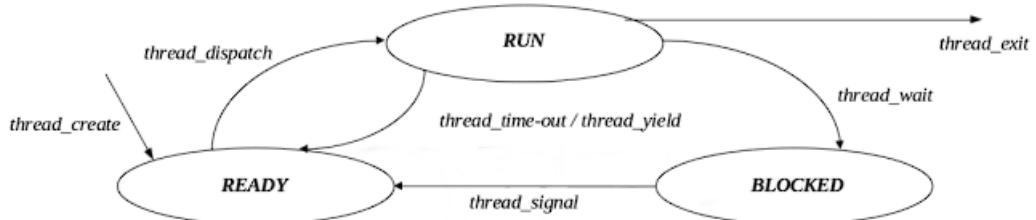


### - Typical Unix state diagram



- Existem dois estados de execução, **kernel em execução** e **utilizador em execução**, associados ao modo de funcionamento do processador, supervisor e utilizador, respetivamente
- O estado pronto também é dividido em dois estados, **pronto para executar em memória** e **preempted**, mas são equivalentes, representados pela linha tracejada
- Quando um processo de utilizador sai do modo supervisor, pode ser preterido (porque um processo de prioridade mais alta está pronto para ser executado)
- Na prática, os processos **prontos a correr em memória** e **preempted** partilham a mesma fila, pelo que são tratados como iguais
- A transição de **time-out** é coberta pela transição de preempção
- Tradicionalmente, a execução em modo supervisor não podia ser interrompida (assim, o UNIX não permite o processamento em tempo real)
- Nas versões atuais, nomeadamente a partir do SVR4, o problema foi resolvido dividindo o código numa sucessão de regiões atómicas entre as quais as estruturas de dados internas estão num estado seguro, permitindo assim a interrupção da execução
- Isto corresponde a uma transição entre os estados **preempted** e **kernel running**, que poderia ser designada por **regresso ao kernel**

### - Threads : State diagram of a thread



- Apenas são considerados os estados relativos à gestão do processador (estados de curto prazo)
- os estados **suspensão-pronto** e **suspensão-bloqueado** não estão presentes:
  - estão relacionados com o processo e não com as threads
- os estados **new** e **terminated** não estão presentes:

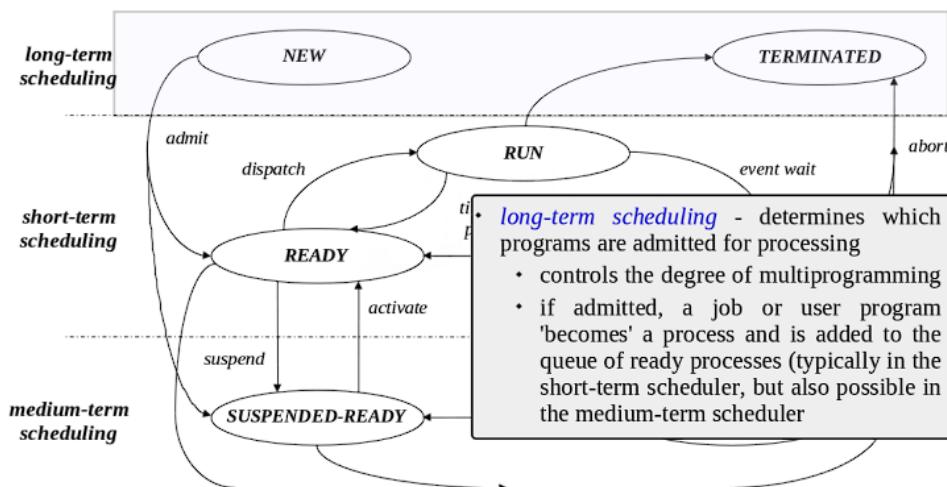
- o a gestão do ambiente de multiprogramação está basicamente relacionada com restringir o número de threads que podem existir num processo

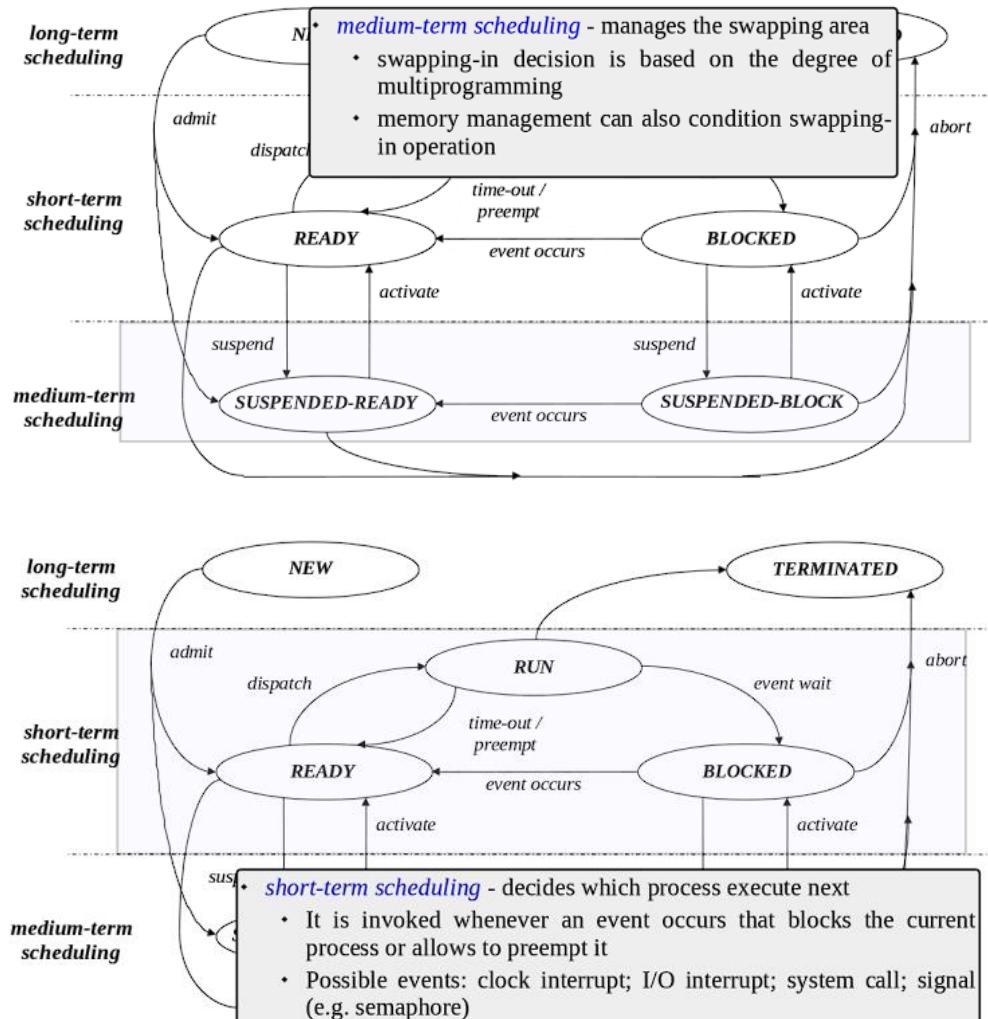
## Processor scheduling

### - Processor scheduler : Definition

- A execução de um processo é, por si só, uma sequência alternada de dois tipos de períodos:
  - o CPU Burst - execução de instruções da CPU
  - o I/O burst - aguardando a conclusão de um pedido de I/O
- Com base nisso, um processo pode ser classificado como:
  - o I/O-bound - se tiver muitos bursts curtos de CPU
  - o CPU-bound (ou processor-bound) - se tiver (poucas) longas CPU bursts
- A ideia por trás da multiprogramação é aproveitar os períodos de I/O burst para colocar outros processos usando a CPU
- O agendador do processador é o componente do sistema operativo responsável por decidir como a CPU é atribuída para a execução dos CPU bursts dos vários processos que coexistem no sistema

### - Processor scheduler : Levels in processor scheduling





### - Short-term processor scheduler : Preemption & non-preemption

- O agendador de curto prazo do processador pode ser **preemptivo** ou **não preemptivo**
- Agendamento não preemptivo** - um processo mantém o processador até bloquear ou terminar
  - As transições **time-out** e **preempt** não existem
  - Típico em sistemas batch.
- Agendamento preemptivo** - um processo pode perder o processador devido a razões externas
  - por esgotamento do quantum de tempo atribuído (transição **time-out**)
  - porque um processo mais prioritário está pronto para ser executado (transição de **preempção**)

### - Scheduling algorithms : Evaluation criteria

- O principal objetivo do agendamento a curto prazo é atribuir tempo ao processador para otimizar uma função objetivo do comportamento do sistema
- Deve ser estabelecido um conjunto de critérios para a avaliação
- Estes critérios podem ser vistos de diferentes perspetivas:
  - Critérios orientados para o utilizador** - relacionados com o comportamento do sistema tal como é percebido pelo utilizador ou processo individual

- Critérios orientados para o sistema - relacionados com a utilização efetiva e eficiente do processador
- Um critério pode ser uma medida quantitativa direta/indireta ou apenas uma qualitativa
- Os critérios de agendamento são interdependentes, pelo que é impossível otimizar todos eles simultaneamente

#### - Scheduling criteria : User-oriented scheduling criteria

- Tempo de execução - intervalo de tempo entre a apresentação de um processo/trabalho e a sua conclusão (inclui o tempo de execução efetivo mais o tempo de espera pelos recursos, incluindo o processador)
  - medida adequada para um batch job
  - deve ser minimizado
- Tempo de espera - soma dos períodos passados por um processo à espera no estado pronto
  - deve ser minimizado
- Tempo de resposta - tempo desde a submissão de um pedido até que a resposta começa a ser recebida
  - medida apropriada para um processo interativo
  - deve ser minimizado
  - mas também o número de processos interativos com um tempo de resposta aceitável também deve ser maximizado
- Prazos - tempo de conclusão de um processo
  - a percentagem de prazos cumpridos deve ser maximizada, mesmo subordinando outros objetivos
- Previsibilidade - como a resposta é afetada pela carga no sistema
  - Um determinado trabalho deve ser executado aproximadamente no mesmo período de tempo e com aproximadamente o mesmo custo independentemente da carga no sistema

#### - Scheduling criteria : System-oriented scheduling criteria

- Equidade - igualdade de tratamento
  - Na ausência de orientações, os processos devem ser tratados da mesma forma, e nenhum processo deve sofrer inanição
- Taxa de transferência (Throughput) - número de processos concluídos por unidade de tempo
  - mede a quantidade de trabalho que está a ser realizado pelo sistema
  - deve ser maximizado
  - depende das durações médias dos processos, mas também da política de agendamento
- Utilização do processador - percentagem de tempo em que o processador está ocupado
  - deve ser maximizada (especialmente em sistemas partilhados dispendiosos)
- Aplicação de prioridades (Enforcing priorities )
  - os processos com maior prioridade devem ser favorecidos
- Como referido anteriormente, é impossível satisfazer todos os critérios em simultâneo

- Os processos a favorecer dependem da aplicação

### - Priorities : Types of priorities

- Priorities can be:

- static – if they do not change over time
- dynamic – if they depend on the past history of execution of the processes

- Static priorities:

- Processes are grouped into fixed priority classes, according to their relative importance
- Clear risk of **starvation** of lower-priority processes
- The most **unfair** discipline
- Typical in real-time systems – **why?**

- Dynamic, deterministically changing priorities:

- When a process is created, a given priority level is assigned to it
- on **time-out** the priority is decremented
- on **event wait** the priority is incremented
- when a minimum value is reached, the priority is set to the initial value

- Dynamic priorities:

- Priority classes are functionally defined
- In interactive systems, change of class can be based on how the last execution window was used
  - level 1 (highest priority): **terminals** – a process enters this class on event occurs if it was waiting for data from the standard input device
  - level 2: **generic I/O** – a process enters this class on event occurs if it was waiting for data from another type of input device
  - level 3: **small time quantum** – a process enters this class on time-out
  - level 4: (lowest priority): **large time quantum** – a process enters this class after a successive number of time-outs.
- They are clearly CPU-bound processes and the idea is given them large execution windows, less times

- Dynamic priorities:

- In batch systems, the turnaround time should be minimized
- If estimates of the execution times of a set of processes are known in advance, it is possible to establish an order for the execution of the processes that minimizes the average turnaround time of the group.
- Assume  $N$  jobs are submitted at time 0, whose estimates of the execution times are  $t_{e_n}$ , with  $n = 1, 2, \dots, N$ .
  - The turnaround time of job  $i$  is given by

$$t_{t_i} = t_{e_1} + t_{e_2} + \dots + t_{e_i}$$

- The average turnaround time is given by

$$t_m = \frac{1}{N} \sum_{i=1}^N t_{t_i} = t_{e_1} + \frac{N-1}{N} t_{e_2} + \dots + \frac{1}{N} t_{e_N}$$

- $t_m$  is **minimum** if jobs are scheduled in ascending order of the estimated execution times

- **Dynamic priorities:**

- An approach similar to the previous one can be used in interactive systems
- The idea is to estimate the occupancy fraction of the next execution window, based on the occupation of the past windows, and assign the processor to the process for which this estimate is the lowest
- Let  $e_1$  be the estimate of the occupancy fraction of the first execution window assigned to a process and let  $f_1$  be the first fraction effectively verified. Then:
  - estimate  $e_2$  is given by

$$e_2 = a.e_1 + (1 - a).f_1 \quad , \quad a \in [0, 1]$$

- estimate  $e_N$  is given by

$$e_N = a.e_{N-1} + (1 - a).f_{N-1} \quad , \quad a \in [0, 1]$$

$$= a^{N-1}.e_1 + a^{N-2}.(1 - a).f_1 + \dots + a.(1 - a).f_{N-2} + (1 - a).f_{N-1}$$

- coefficient  $a$  is used to control how much the past history of execution influences the present estimate

- In the previous approach, CPU-bound processes can suffer **starvation**
- To overcome that problem, the **aging** of a process in the READY queue can be part of the equation
- Let  $R$  be such time, typically normalized in terms of the duration of the execution interval.
- Then, priority  $p$  of a process can be given by

$$p = \frac{1 + b.R}{e_N}$$

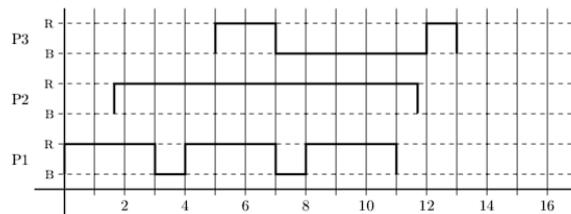
where  $b$  is a coefficient that controls how much the aging weights in the priority

### - Scheduling policies : FCFS

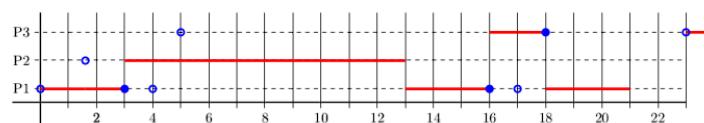
- **First-Come-First-Server (FCFS)** scheduler

- Also known as **First-In-First-Out (FIFO)**
- The oldest process in the READY queue is the first to be selected
- Non-preemptive (in strict sense)
  - Can be **combined with a priority schema** (in which case preemption could exist)
- Favours CPU-bound processes over I/O-bound
- Can result in bad use of both processor and I/O devices

### Exemplo:



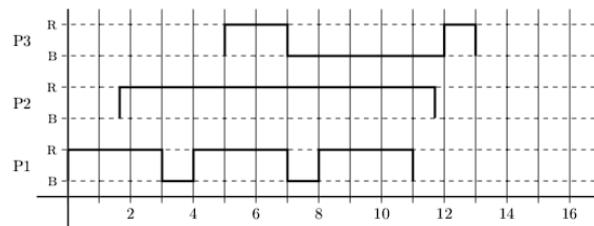
- Draw processor utilization, assuming FCFS policy and no priorities



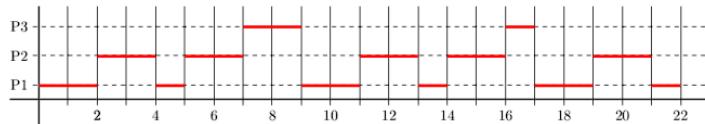
### - Scheduling policies : RR

- Round robin (RR) scheduler
  - Preemptive (base on a clock)
    - Each process is given a maximum slice of time before being preempted (**time quantum**)
    - Also known as **time slicing**
  - The oldest process in the READY queue is the first one to be selected (no priorities)
    - Can be combined with a priority schema
  - The principal design issue is the time quantum
    - very short is good, because short processes will move through the system quickly and response time is minimized
    - very short is bad, because every context switching involves a processing overhead
  - Effective in general purpose time-sharing systems and in transaction processing systems
  - Favours CPU-bound processes over I/O-bound
  - Can result in bad use of I/O devices

**Exemplo:**



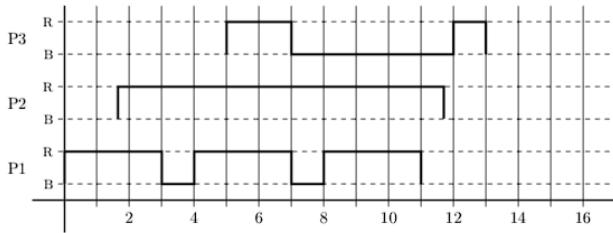
- Draw processor utilization, assuming RR policy with a time quantum of 2 and no priorities



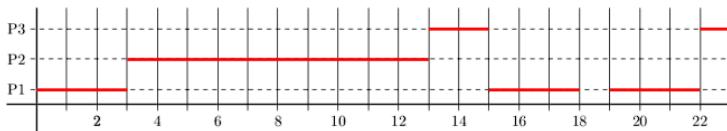
**- Scheduling policies : SPN**

- Shortest Process Next (SPN) scheduler
  - Also known as shortest job first (SJF)
  - Non-preemptive
  - The process with the **shortest expected next CPU burst time** is selected next
    - FCFS is used to tie up (in case several processes have the same burst time)
  - Maximum throughput
  - Minimum average waiting time and turnaround time
  - Risk of **starvation** for longer processes
  - Requires the knowledge in advance of the (**expected**) processing time
    - This value can be predicted, using the previous values
  - Used in long-term scheduling in batch systems
    - where users are motivated to estimate the process time limit accurately

### Exemplo:



- Draw processor utilization, assuming SPN policy and no priorities



### Interprocess communication

#### - Access primitives : Requirements

- Requisitos que devem ser observados no acesso a uma secção crítica:
  - Exclusão mútua efetiva - o acesso às secções críticas associadas ao um mesmo recurso, ou área partilhada, só pode ser permitido a um processo de cada vez, entre todos os processos que competem pelo acesso
  - Independência do número de processos intervenientes ou da sua velocidade relativa de execução
  - um processo fora da sua secção crítica não pode impedir outro processo de entrar na sua própria secção crítica
  - Sem fome - um processo que necessite de acesso à sua secção crítica não deve ter de esperar indefinidamente
  - O tempo de permanência numa secção crítica deve ser necessariamente finito

#### - Access primitives : Types of solutions

- Em geral, uma localização de memória é utilizada para controlar o acesso à secção crítica
  - funciona como um sinalizador binário
- Dois tipos de soluções: soluções de software e soluções de hardware
- soluções de software - soluções que se baseiam nas instruções típicas utilizadas para aceder à posição de memória
  - a leitura e a escrita são efetuadas por instruções diferentes
  - pode ocorrer uma interrupção entre a leitura e a escrita
- soluções de hardware - soluções que se baseiam em instruções especiais para aceder à posição de memória
  - estas instruções permitem ler e depois escrever uma posição de memória de forma atómica (sem interrupções)

### - Software solutions : Dekker algorithm (1965)

```
#define R 2 /* process id = 0, 1 */
shared bool want_enter[R] = {false, false};
shared uint p_w_priority = 0;
void enter_critical_section(uint own_pid)
{
    uint other_pid = 1 - own_pid;
    want_enter[own_pid] = true;
    while (want_enter[other_pid])
    {
        if (own_pid != p_w_priority)
        {
            want_enter[own_pid] = false;
            while (own_pid != p_w_priority);
            want_enter[own_pid] = true;
        }
    }
    void leave_critical_section(uint own_pid)
    {
        uint other_pid = 1 - own_pid;
        p_w_priority = other_pid;
        want_enter[own_pid] = false;
    }
}
```

- The algorithm uses an alternation mechanism (on the priority) to solve the conflict
- Mutual exclusion in the access to the critical section is guaranteed
- Deadlock and starvation are not present
- No assumptions are done in the relative speed of the intervening processes
- However, it can **not be generalized** to more than 2 processes, satisfying all the requirements

### - Software solutions Dijkstra algorithm (1966)

```
#define R ... /* process id = 0, 1, ..., R-1 */
shared uint want_enter[R] = {NO, NO, ... , NO};
shared uint p_w_priority = 0;
void enter_critical_section(uint own_pid)
{
    uint n;
    do
    {
        want_enter[own_pid] = WANT;
        while (own_pid != p_w_priority)
            if (want_enter[p_w_priority] == NO)
                p_w_priority = own_pid;
        want_enter[own_pid] = DECIDED;
        for (n = 0; n < R; n++)
            if (n != own_pid && want_enter[n] == DECIDED)
                break;
    } while (n < R);
}
void leave_critical_section(uint own_pid)
{
    p_w_priority = (own_pid + 1) % R;
    want_enter[own_pid] = NO;
}
```

- Works, but can suffer from **starvation**

### - Software solutions : Peterson algorithm (1981)

```

#define R 2 /* process id = 0, 1 */
shared bool want_enter[R] = {false, false};
shared uint last;
void enter_critical_section(uint own_pid)
{
    uint other_pid = 1 - own_pid;
    want_enter[own_pid] = true;
    last = other_pid;
    while ((want_enter[other_pid]) && (last == other_pid));
}
void leave_critical_section(uint own_pid)
{
    want_enter[own_pid] = false;
}

```

- The Peterson algorithm uses the order of arrival to solve conflicts
- Each process has to write the other's ID in a shared variable (last)
- The subsequent reading allows to determine which was the last one

- The Peterson algorithm uses the order of arrival to solve conflicts
  - Each process has to write the other's ID in a shared variable (last)
  - The subsequent reading allows to determine which was the last one
- It is a valid solution
  - Guarantees mutual exclusion
  - Avoids deadlock and starvation
  - Makes no assumption about the relative speed of intervening processes

### - Generalized Peterson algorithm (1981)

```

#define R ... /* process id = 0, 1, ..., R-1 */
shared int level[R] = {-1, -1, ..., -1};
shared int last[R-1];
void enter_critical_section(uint own_pid)
{
    for (uint i = 0; i < R-1; i++)
    {
        level[own_pid] = i;
        last[i] = own_pid;
        do
        {
            test = false;
            for (uint j = 0; j < R; j++)
                if (j != own_pid)
                    test = test || (level[j] >= i);
        } while (test && (last[i] == own_pid));
    }
}
void leave_critical_section(int own_pid)
{
    level[own_pid] = -1;
}

```

- Can be generalized to more than two processes
- The general solution is similar to a waiting queue

### - Hardware solutions : disabling interrupts

- Sistema computacional uniprocessador
  - A comutação de processos, num ambiente multiprogramado, é sempre causada por um dispositivo externo:
    - relógio de tempo real (RTC) - causa a transição de time-out em sistemas preemptivos
    - controlador de dispositivo - pode provocar transições de preempção em caso de despertar de um processo de prioridade mais elevada
    - Em qualquer caso, interrupções do processador
  - Assim, o acesso em exclusão mútua pode ser implementado desativando as interrupções
  - Válido apenas no kernel

- O código malicioso ou com erros pode bloquear completamente o sistema
- Sistema computacional multiprocessador
  - A desativação das interrupções num processador não tem qualquer efeito

#### - Hardware solutions : special instructions – TAS

```

shared bool flag = false;

bool test_and_set(bool * flag)
{
    bool prev = *flag;
    *flag = true;
    return prev;
}

void lock(bool * flag)
{
    while (test_and_set(flag));
}

void unlock(bool * flag)
{
    *flag = false;
}

```

- The `test_and_set` function, if implemented **atomically** (without interruptions), can be used to construct the `lock` (`enter critical section`) primitive
- In the instruction set of some of the current processors, there is an atomic instruction implementing this behavior
- Surprisingly, it is often called **TAS (test and set)**

#### - Hardware solutions special instructions – CAS

```

shared int value = 0;

int compare_and_swap(int * value,
                     int expected, int new_value)
{
    int v = *value;
    if (*value == expected)
        *value = new_value;
    return v;
}

void lock(int * flag)
{
    while (compare_and_swap(&flag,
                           0, 1) != 0);
}

void unlock(bool * flag)
{
    *flag = 0;
}

```

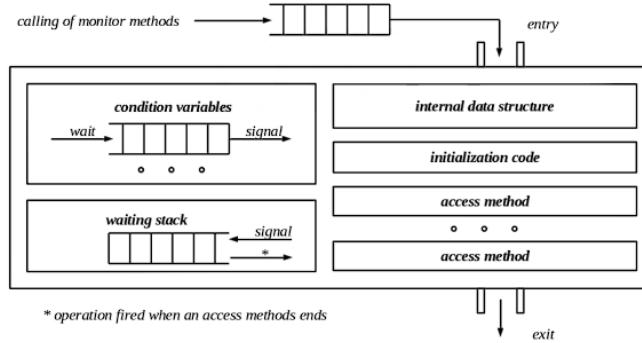
- The `compare_and_swap` function, if implemented **atomically** (without interruptions), can be used to construct the `lock` (`enter critical section`) primitive
- In the instruction set of some of the current processors, there is an atomic instruction implementing that behavior
- In some instruction sets, there is a `compare_and_set` variant this returns a bool

#### - Hardware solutions : Busy waiting

- As soluções anteriores sofrem de **espera ocupada (busy waiting)**
  - A primitiva de **bloqueio** está no estado ativo (utilizando a CPU) enquanto espera
  - É frequentemente referido como um **spinlock**, pois o processo gira em torno da variável enquanto espera pelo acesso
- Em **sistemas uniprocessadores**, a espera ocupada é indesejada, pois há
  - **perda de eficiência** - o quantum de tempo de um processo é utilizado para nada
  - **risco de impasse** - se um processo de prioridade mais elevada chamar o bloqueio enquanto um processo de prioridade mais baixa está dentro da sua secção crítica, nenhum deles pode prosseguir
- Em **sistemas multiprocessadores** com memória partilhada, a espera ocupada pode ser menos crítica
  - a troca de processos custa tempo, que pode ser maior do que o tempo gasto pelo outro processo dentro da sua secção crítica

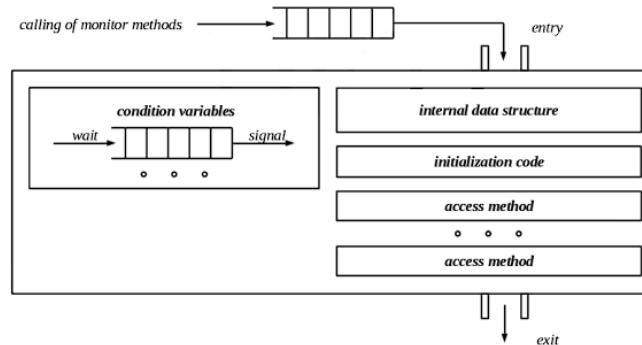
## - Monitors : Hoare monitor

- **Hoare monitor** – the thread calling signal is put out of the monitor, so the just waked up thread can proceed
  - quite general, but its implementation requires a stack where the blocked thread is put



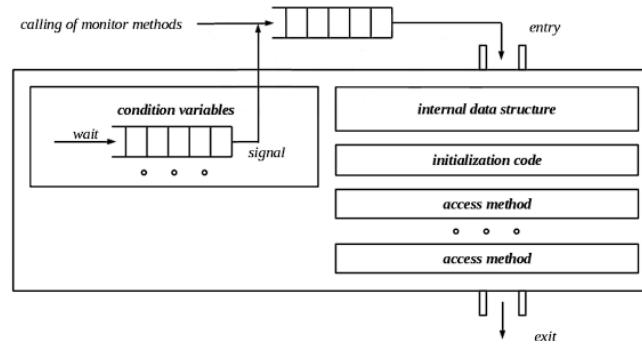
## - Monitors : Brinch Hansen monitor

- **Brinch Hansen monitor** – the thread calling signal immediately leaves the monitor (signal is the last instruction of the monitor method)
  - easy to implement, but quite restrictive (only one signal allowed in a method)



## - Monitors : Lampson / Redell monitor

- **Lampson / Redell monitor** – the thread calling signal continues its execution and the just waked up thread is kept outside the monitor, competing for access
  - easy to implement, but can cause starvation



## Message-passing

### Introduction

- Processes can communicate exchanging messages
  - A general communication mechanism, not requiring explicit shared memory, that includes both communication and synchronization
  - Valid for uniprocessor and multiprocessor systems
- Two operations are required:
  - `send` and `receive`
- A communication link is required
  - That can be categorized in different ways:
    - Direct or indirect communication
    - Synchronous or asynchronous communication
    - Type of buffering

## Message-passing

### Direct and indirect communication

- **Symmetric direct communication**
  - A process that wants to communicate must explicitly name the receiver or sender
    - `send(P, msg)` – send message `msg` to process `P`
    - `receive(P, msg)` – receive message `msg` from process `P`
  - A communication link in this scheme has the following properties:
    - it is established automatically between a pair of communicating processes
    - it is associated with exactly two processes
    - between a pair of communicating processes there exist exactly one link
- **Asymmetric direct communication**
  - Only the sender must explicitly name the receiver
    - `send(P, msg)` – send message `msg` to process `P`
    - `receive(id, msg)` – receive message `msg` from any process

## Message-passing

### Direct and indirect communication

- **Indirect communication**
  - The messages are sent to and received from mailboxes, or ports
    - `send(M, msg)` – send message `msg` to mailbox `M`
    - `receive(M, msg)` – receive message `msg` from mailbox `M`
  - A communication link in this scheme has the following properties:
    - it is only established if the pair of communicating processes has a shared mailbox
    - it may be associated with more than two processes
    - between a pair of processes there may exist more than one link (a mailbox per each)
  - The problem of two or more processes trying to receive a message from the same mailbox
    - Is it allowed?
    - If allowed, which one will succeed?

## Message-passing Synchronization

- From a synchronization point of view, there are different design options for implementing `send` and `receive`
  - **Blocking send**– the sending process blocks until the message is received by the receiving process or by the mailbox
  - **Nonblocking send**– the sending process sends the message and resumes operation
  - **Blocking receive**– the receiver blocks until a message is available
  - **Nonblocking receive**– the receiver retrieves either a valid message or the indication that no one exists
- Different combinations of send and receive are possible

## Message-passing Buffering

- There are different design options for implementing the link supporting the communication
  - **Zero capacity** – there is no queue
    - the sender must block until the recipient receives the message
  - **Bounded capacity** – the queue has finite length
    - if the queue is full, the sender must block until space is available
  - **Unbounded capacity** – the queue has (potentially) infinite length

## Unix IPC primitives Message-passing

- **System V implementation**
  - Defines a message queue where messages of different types (a positive integer) can be stored
  - The send operation blocks if space is not available
  - The receive operation has an argument to specify the type of message to receive: a given type, any type or a range of types
    - The oldest message of given type(s) is retrieved
    - Can be blocking or nonblocking
  - see system calls: `msgget`, `msgsnd`, `msgrcv`, and `msgctl`
- **POSIX message queue**
  - Defines a priority queue
  - The send operation blocks if space is not available
  - The receive operation removes the oldest message with the highest priority
    - Can be blocking or nonblocking
  - see functions: `mq_open`, `mq_send`, `mq_receive`, ...

## Deadlock

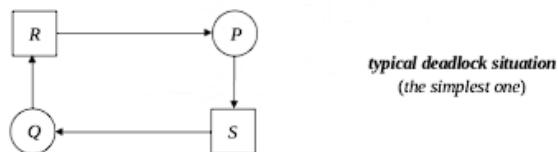
### - Deadlock : Introduction

- De um modo geral, um **recurso** é algo de que um processo necessita para prosseguir a sua execução
  - **componentes físicos do sistema computacional** (processador, memória, dispositivos de I/O, etc.)
  - **estruturas de dados comuns** definidas a nível do sistema operativo (PCT, canais de comunicação, etc.) ou entre processos de uma dada aplicação
- Os recursos podem ser:
  - **preemptivos** - se puderem ser retirados dos processos que os detêm
    - ex: processador, regiões de memória utilizadas pelo espaço de endereçamento de um processo

- **não preemptivos** - se só puderem ser libertados pelos processos que os detêm
  - ex: um ficheiro, uma região de memória partilhada que requer acesso exclusivo para a sua manipulação

#### - Deadlock : Necessary conditions for deadlock

- Pode provar-se que, quando ocorre um deadlock, 4 condições são necessariamente observadas:
  - **exclusão mútua** - apenas um processo pode utilizar um recurso de cada vez
    - se outro processo o solicitar, deve esperar até que seja libertado
  - **hold and wait** - Um processo mantém recursos em sua posse enquanto espera por outro
  - **não preempção** - os recursos não podem ser preemitidos
    - apenas o processo que detém um recurso pode libertá-lo (provavelmente depois de completar a tarefa que o requer)
  - **espera circular** - deve existir um conjunto de processos em espera de tal forma que cada um deles esteja à espera para recursos detidos por outros processos no conjunto
    - existem loops no grafo



#### - Deadlock prevention : Denying the necessary conditions

- A negação da condição de **exclusão mútua** só é possível se os recursos forem partilháveis ao mesmo tempo
  - Caso contrário, podem ocorrer condições de corrida
- A negação da condição de **preempção** só é possível se os recursos forem preemptivos
  - O que muitas vezes não é o caso
- Assim, em geral, apenas as outras condições (**hold-and-wait** e **espera circular**) são usadas para implementar a prevenção de deadlock
- Evitar a condição de **hold-and-wait** pode ser feito se um processo solicitar todos os recursos necessários de uma só vez
  - Nesta solução, pode ocorrer fome (starvation)
  - Os mecanismos de **envelhecimento** são frequentemente utilizados para resolver a fome
- A negação da condição de **hold-and-wait** também pode ser efetuada se um processo libertar o(s) recurso(s) já adquirido(s) se não conseguir adquirir o próximo
- Neste tipo de soluções, pode ocorrer fome e busy waiting
  - Para evitar busy waiting o processo deve bloquear e ser acordado quando o recurso for libertado
- A negação da condição de **espera circular** pode ser efetuada atribuindo um id numérico diferente a cada recurso e impondo que a aquisição de recursos tenha de ser feita por ordem ascendente ou descendente
  - Desta forma, a cadeia circular é sempre evitada
  - A fome não é evitada

### - Deadlock avoidance : Definition

- A deadlock avoidance é menos restritiva do que a prevenção de deadlock
  - As condições de deadlock não são negadas pelo lado da aplicação
  - Existe um gestor de recursos que decide o que fazer em termos de alocação de recursos
  - Requer conhecimento prévio dos pedidos máximos de recursos dos processos
    - Os processos intervenientes têm de declarar no início as suas necessidades máximas em termos de recursos
- Duas abordagens possíveis
  - Negação de início de processo
    - Não iniciar um processo se os seus pedidos puderem conduzir a um deadlock
  - Negação de atribuição de recursos
    - Não conceder um pedido de recurso incremental a um processo se essa atribuição puder conduzir a deadlock

### Deadlock avoidance

Process initiation denial

- The system prevents a new process to start if its termination can not be guaranteed
- Let
  - $R = (R_1, R_2, \dots, R_n)$  be a vector of the total amount of each resource
  - $P$  be the set of processes competing for resources
  - $C_p$  be a vector of the total amount of each resource declared by process  $p \in P$
- A new process  $q$  ( $q \notin P$ ) is only started if

$$C_q \leq R - \sum_{p \in P} C_p$$

- It is a quite restrictive approach

### Deadlock avoidance

Resource allocation denial

- A new resource is allocated to a process if and only if there is at least one sequence of future allocations that does not result in deadlock
  - In such cases, the system is said to be in a **safe state**
- Let
  - $R = (R_1, R_2, \dots, R_n)$  be a vector of the total amount of each resource
  - $V = (V_1, V_2, \dots, V_n)$  be a vector of the amount of each resource available
  - $P$  be the set of processes competing for resources
  - $C_p$  be a vector of the total amount of each resource declared by process  $p \in P$
  - $A_p$  be a vector of the amount of each resource already allocated to process  $p \in P$
- A new request of a process  $q$  is only granted if, after it, there is a sequence  $s(k)$ , with  $s(k) \in P$  and  $k = 1, 2, \dots, |P|$ , of processes, such that

$$C_{s(k)} - A_{s(k)} = V + \sum_{m=1}^{k-1} A_{s(m)}$$

- This approach is called the **banker's algorithm**

### - Deadlock detection : Definition

- Não é utilizada qualquer prevenção ou evitação de deadlock
  - Assim, podem ocorrer situações de deadlock

- O estado do sistema deve ser examinado para determinar se ocorreu um deadlock
- Deve existir e ser aplicado um procedimento de recuperação do deadlock

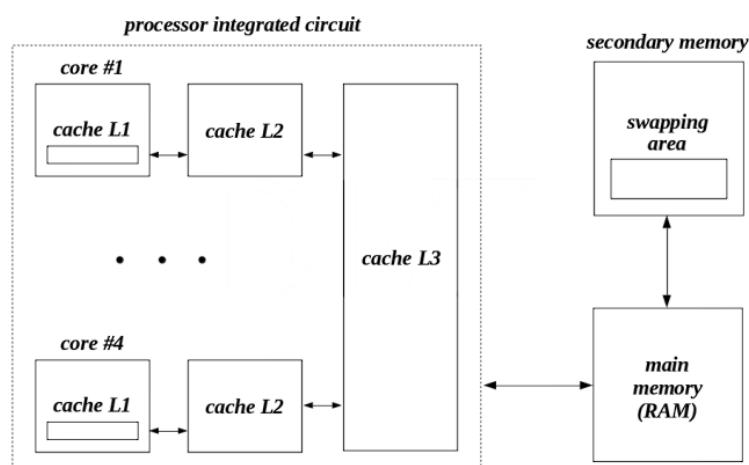
#### - Deadlock detection : Recover procedure

- Como?
  - libertar recursos de um processo - se for possível
    - O processo é suspenso até que o recurso possa ser devolvido
    - Eficiente, mas requer a possibilidade de guardar o estado do processo
  - reversão - se os estados de execução dos diferentes processos forem guardados periodicamente
    - Um recurso é libertado de um processo, cujo estado de execução é revertido para o momento em que o recurso foi atribuído a ele
  - matar processos
    - Método radical, mas fácil de implementar

#### Memory management

#### - Memory management : Memory hierarchy

- Idealmente, um programador de aplicações gostaria de ter uma memória disponível infinitamente grande, infinitamente rápida, não volátil e barata
  - Na prática, tal não é possível
- Assim, a memória de um sistema informático está normalmente organizada em diferentes níveis, formando uma hierarquia
  - memória cache - pequena (dezenas de KB a alguns MB), muito rápida, volátil e cara
  - memória principal - tamanho médio (centenas de MB a centenas de GB), volátil e preço médio e velocidade de acesso média
  - memória secundária - grande (dezenas, centenas ou milhares de GB), lenta, não volátil e barata



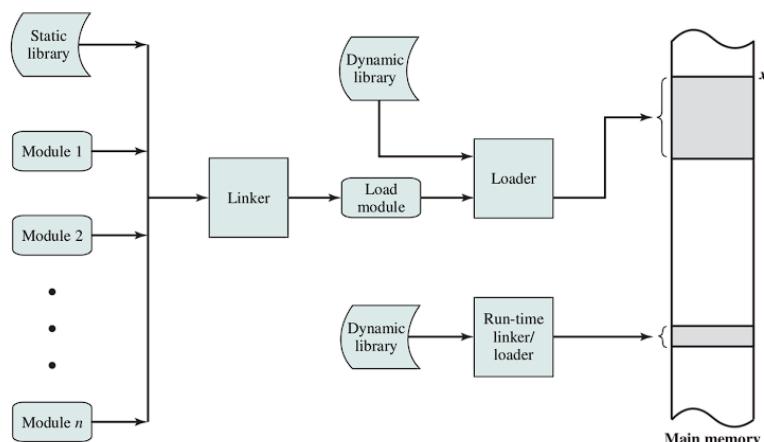
- A memória cache conterá uma cópia das posições de memória (instruções e operandos) mais frequentemente referenciadas pelo processador num passado próximo
  - A memória cache está localizada no próprio circuito integrado do processador (nível 1)

- É num circuito integrado autónomo colado no mesmo substrato (níveis 2 e 3)
- A transferência de dados de e para a memória principal é efetuada de forma quase totalmente transparente para o programador do sistema
- A **memória secundária** tem duas funções principais
  - **Sistema de ficheiros** - armazenamento de informação mais ou menos permanente (programas e dados)
  - **Área de troca** - Extensão da memória principal para que o seu tamanho não constitua um fator limitativo do número de processos que podem coexistir atualmente
    - a área de troca pode estar numa partição de disco utilizada apenas para esse fim ou ser um ficheiro num sistema de ficheiros
- Este tipo de organização baseia-se no pressuposto de que quanto mais longe uma instrução ou operando estiver longe do processador, menos vezes ele será referenciado
  - Nestas condições, o tempo médio de uma referência tende a aproximar-se do valor mais baixo
- Baseado no **princípio da localidade de referência**
  - A tendência de um programa para aceder repetidamente ao mesmo conjunto de posições de memória durante um curto período de tempo

#### - Memory management : Role

- O papel da gestão da memória num ambiente de multiprogramação centra-se na atribuição de memória aos processos e no controlo da transferência de dados entre a memória principal e a secundária (**área de troca**), de modo a
  - **Manter um registo** das partes da memória principal que estão ocupadas e as que estão livres
  - **Reservar porções** da memória principal para os processos que vão precisar delas, ou libertá-las quando já não são necessárias
  - **Swapping out** todo ou parte do espaço de endereço de um processo quando a memória principal é muito pequena para conter todos os processos que coexistem.
  - **Swapping in** todo ou parte do espaço de endereço de um processo quando a memória principal fica disponível

#### - Address space : Linker and loader roles



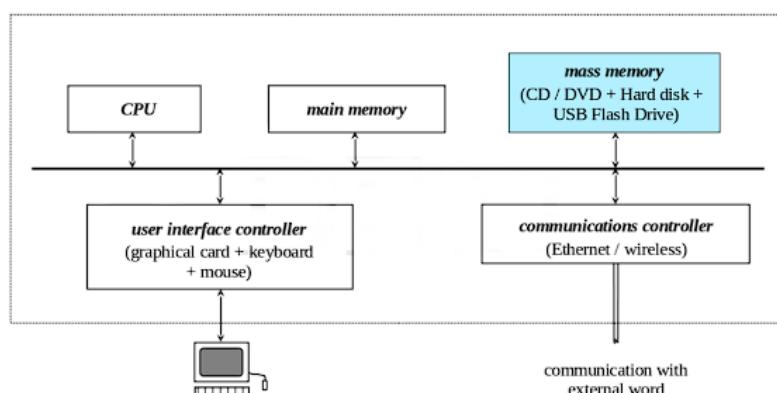
- Os ficheiros-objeto, resultantes do processo de compilação, são ficheiros relocalizáveis
  - Os endereços das várias instruções, constantes e variáveis são calculados a partir do início do módulo, por convenção o endereço 0
- O papel do processo de ligação é reunir os diferentes ficheiros objeto num único ficheiro, o ficheiro executável, resolvendo entre si as várias referências externas
  - As bibliotecas estáticas também são incluídas no ficheiro executável
  - As bibliotecas dinâmicas (partilhadas) não o são
- O carregador constrói a imagem binária do espaço de endereço do processo, que eventualmente será executada, combinando o ficheiro executável e, se aplicável, algumas bibliotecas dinâmicas, resolvendo quaisquer referências externas restantes
- As bibliotecas dinâmicas também só podem ser carregadas em tempo de execução
- Quando a ligação é dinâmica
  - Cada referência no código a uma rotina de uma biblioteca dinâmica é substituída por um stub
    - um pequeno conjunto de instruções que determina a localização de uma rotina específica, se ela já estiver residente na memória principal, ou promove o seu carregamento na memória, caso contrário
  - Quando um stub é executado, a rotina associada é identificada e localizada na memória principal, o stub substitui então a referência ao seu endereço no código de processo pelo endereço da rotina de sistema e executa-a
  - Quando essa zona de código é alcançada novamente, a rotina do sistema é agora executada diretamente
- Todos os processos que usam a mesma biblioteca dinâmica executam a mesma cópia do código, minimizando assim a ocupação da memória principal

## VER APRESENTAÇÃO MEMORY MANAGEMENT APARTIR DO SLIDE 7

## VER APRESENTAÇÃO FILE SYSTEMS

### File systems

#### - Overview : The mass storage



#### - Overview : Importance of mass storage (secondary memory)

- Armazenamento do sistema operativo
  - Quando um sistema informático é ligado, existe apenas um programa na memória principal (uma pequena região semelhante a uma ROM), o gestor de arranque, cuja principal função é ler de uma região específica da memória

de massa um programa maior que carrega na memória principal, e executa, o programa que implementa o ambiente de interação com o utilizador ambiente

- **Armazém de aplicações**
  - Para que um sistema informático possa efetuar um trabalho útil, deve existir um local permanente onde armazenar as diferentes aplicações deve existir
- **Armazém de ficheiros do utilizador**
  - Além disso, quase todos os programas, durante a sua execução, produzem, consultam e/ou alteram quantidades variáveis de informação que, de forma mais ou menos permanente, devem ser armazenadas

#### - Overview : Properties of mass storage

- **non-volatility** – information exists beyond the processes that produce and/or use it, even after the computer is turned off
  - **large storage capacity** – the information manipulated by the computer processes can far exceed that which is directly stored in their own address spaces
  - **accessibility** – access to stored information should be done in the simplest and fastest way possible
  - **integrity** – the stored information must be protected against accidental or malicious corruption
  - **sharing of information** – the information must be accessible concurrently to the multiple processes that make use of it
- 
- **File system** is the part of the operating system responsible to manage access to mass storage contents

#### - File concept : What is a file?

- **file** is the **logical unit of storage** in mass storage
  - meaning that reading and writing information is always done within the strict scope of a file
  - But have in mind that the **physical unit of interaction** is the block
- Basic elements of a file:
  - **name/path** – the user (generic) way of referring to the information
  - **identity card** – meta-data (ownership, size, permissions, times, ...)
  - **contents** – the information itself, organized as a sequence of bits, bytes, lines or registers, whose precise format is defined by the creator of the file and which has to be known by whoever accesses it
- From the point of view of the application programmer, a file is understood as an **abstract data type**, characterized by a set of **attributes** and a set of **operations**

#### - File concept : Types of files

- From the **operating system point of view**, there are different types of files:
  - **ordinary/regular file** – file whose contents is of the user responsibility
    - from the operating system point of view it is just a sequence of bytes
  - **directory** – file used to track, organize and locate other files and directories
  - **shortcut (symbolic link)** – file that contains a reference to another file (of any type) in the form of an absolute or relative path
  - **character device** – file representing a device handled in bytes
  - **block device** – file representing a device handled in blocks
  - **socket** – file used for inter-process and inter-machine communication
  - **(named) pipe** – file used for inter-process communication
- Note that text files, image files, video files, application files, etc., are all **regular files**

## - File concept . Attributes of files

- Common attributes of a file
  - **type** – one of the referred above
  - **name/path** – the way users usually refer to the file
  - **internal identification** – the way the file is known internally
  - **size(s)** – size in bytes of information; space occupied on disk
  - **ownership** – who the file belongs to
  - **permissions** – who can access the file and how
  - **access and modification times** – when the file was last accessed or modified
  - **location of data in disk** – ordered set of blocks/clusters of the disk where the file contents is stored

## - File concept : Operations on files

- Common operations on regular files
  - **creation, deletion**
  - **opening, closing** – direct access is not allowed
  - **reading, writing, resizing**
  - **positioning** – in order to allow random access
- Common operations on directories
  - **creation, deletion** (if empty)
  - **opening** (only for reading), **closing**
  - **reading** (directory entries)
    - A directory can be seen as a set/sequence of (directory) entries, each one representing a file (of any valid type)
- Common operations on shortcuts (symbolic links)
  - **creation, deletion**
  - **reading** (the value of the symbolic link)
- Common operations on files of any type
  - **get attributes** (access and modification times, ownership, permissions)
  - **change attributes** (access and modification times, permissions)
  - **change ownership** (only root or admin)

## - Directories : Concept

- Common disks may contain thousands or millions of files
  - It would be impractical to have all that files at the same access level
- **Directory** is a mean to allow the access to disk contents in a hierarchical way
- From a user point of view, a directory can be seen as a container containing files and other directories
- From an implementation point of view, a directory can be seen as a table of directory entries
- Every **directory entry** is a key-value pair that directly or indirectly associates the **name** of a file to its **attributes**

## - Directories : Name and path

- The existence of a file hierarchy makes the **name** insufficient to reference a file in a disk
  - Different files in the hierarchy can exist having the same name
  - How to access a file giving just its name? Not easy, if possible
- The notion of **path** must be introduced
  - A **path** is a sequence of names where all but the last must be directory names or shortcuts pointing to directories
    - In **Unix**, character / is used as separator
    - In **Windows**, character \ is used as separator
    - Names . and .. have special meanings
  - A path can be **absolute** or **relative**
    - An **absolute path** references the location of a file from a root point
    - A **relative path** references the location of a file from an intermediate point
  - In **Unix**, the root point is a single, global root file hierarchy
    - Different storage devices are **mounted** somewhere in this hierarchy
  - In **Windows**, there is a root point per storage device (A:, B:, ...)

### - Data blocks : Some points

- The **block (cluster in Windows)** is the unit of allocation for file contents
  - A block can be a single disk sector (the disk storage unit) or a contiguous sequence of sectors, usually in powers of 2
- Blocks are not shareable among files
  - in general, an in-use block belongs to a single file
- The number of blocks required by a file to store its information is given by
$$N_b = \text{roundup}\left(\frac{\text{size}}{\text{BlockSize}}\right)$$
  - $N_b$  can be **very big** – if block size is 1024 bytes, a 2 GByte file needs 2 MBlocks
  - $N_b$  can be **very small** – a 0-bytes file needs no blocks for data
- It is impractical that all the blocks used by a file are contiguous in disk
- Also, the access to the file data is in general not sequential, but random instead
- So a **flexible data structure**, both in size and location, is required

### - Inodes : What is an inode?

- In Unix, the **inode** (identification node) plays a central role in the implementation of the file data type
  - An inode is typically identified by an integer number
- It corresponds to the **identity card** of a file and contains:
  - file type
  - owner information
  - file access permissions
  - access times
  - file size (in bytes and blocks)
  - sequence of disk blocks with the file contents
- The **name/path** is not in the inode – it is in the directory entry
- In an ext2 file system, in every block group, there is a region reserved for inodes, the **inode table**
- There is also an **inode bitmap** representing the free/allocated inodes