RESUMO SO

Conteudos que saem no Exame teorico

Processes in Unix/Linux

Programas vs Processos

Programa - Set de instruções que descrevem como uma tarefa é executada por um pc.

- Para que a tarefa seja feita, o programa correspondente tem que ser executado.

Processo - Entidade que representa um programa a ser executado.

- Representa uma atividade e é caracterizado por atributos, que vamos ver mais à frente.

Diferentes processos podem estar a correr o mesmo programa.
Em geral, há **mais** processos do que processadores - ***multiprogramming***

Multiprocessing vs. Multiprogramming

Multiprocessing

Paralelismo - habilidade de um sistema computacional de simultaneamente correr 2 ou mais programas. **Mais do que um processador é necessário, cada um para cada execução simultânea)**.

Os sistemas operativos destes sistemas computacionais suportam **multiprocessing**.

Multiprogramming

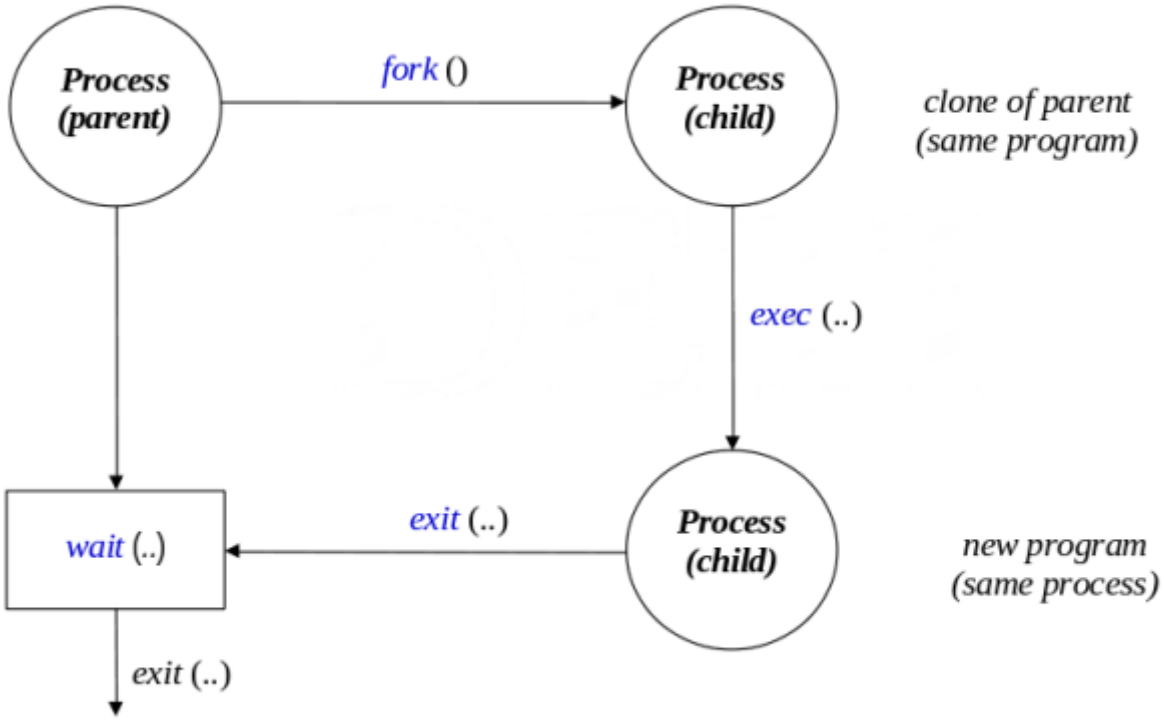
Concorrência - **ILUSÃO** criada por um sistema computacional em que aparentemente, consegue correr varios programas simultâneamente, mais que o número de cpus existentes.

- O processador existente deve estar assigned a varios programas numa *time multiplexed way*

Os sistemas operativos destes sistemas computacionais suportam **multiprogramming**.

Portanto, isto é do tipo, faço um bocado do processo A, o OS faz context switching para o processo B, faço um bocado do processo B...

Criação de processos em UNIX (by cloning)



- O `fork` clona o processo, criando uma replica do mesmo.
- As **address spaces** dos dois processos são iguais. (copy on write).
- Os estados de execução são os mesmos, incluindo o program counter.
- Algumas variaveis do processo sao diferentes (PID, PPID...)

O `fork()` retorna 0 no filho e um valor positivo no pai. Negativo, se sem sucesso. O PPID do filho será o PID do pai.

```
if (fork() == 0) { // criamos um filho aqui
 // é o filho a executar este pedaço de código
 // porque o filho vê o valor retornado do fork como 0.
} else {
 // é o pai a executar este bloco.
}
```

Em geral, queremos **correr um programa diferente no filho**, usando a **system call** `exec`.
Às vezes, queremos que o pai espere pela conclusão de execução do programa do filho.
Para tal, utiliza-se a **system call** `wait`.

Exemplo de código em que se cria filho e o metemos a correr um programa diferente e esperamos pela conclusão.

```
int pid = fork();

if (pid != 0) { // so corre no pai
        int status;
        while (wait(&status) == -1); // esperamos que acabe a execuçao
do filho
        printf(”Process %d (child of %d) ends with status %d\n, pid,
getpid(), WEXITSTATUS (status));
} else { // o filho corre aqui
        execl(aplic, aplic, NULL);
        perror("Fail launching program");
        return EXIT_FAILURE;
}
return EXIT_SUCCESS;
```

Também é possível registar funções para serem chamadas no fim da execução normal de um programa. São chamadas em ordem reversa relativamente ao seu registo.

```
/* cleaning functions */

static void atexit1(void) { printf("atexit1\n"); }

static void atexit2(void) { printf("atexit2\n"); }

/* main programa */

int main(void) {

    /*registering atexit functions*/

    assert(atexit(atexit1) == 0);
    assert(atexit(atexit2) == 0);
```

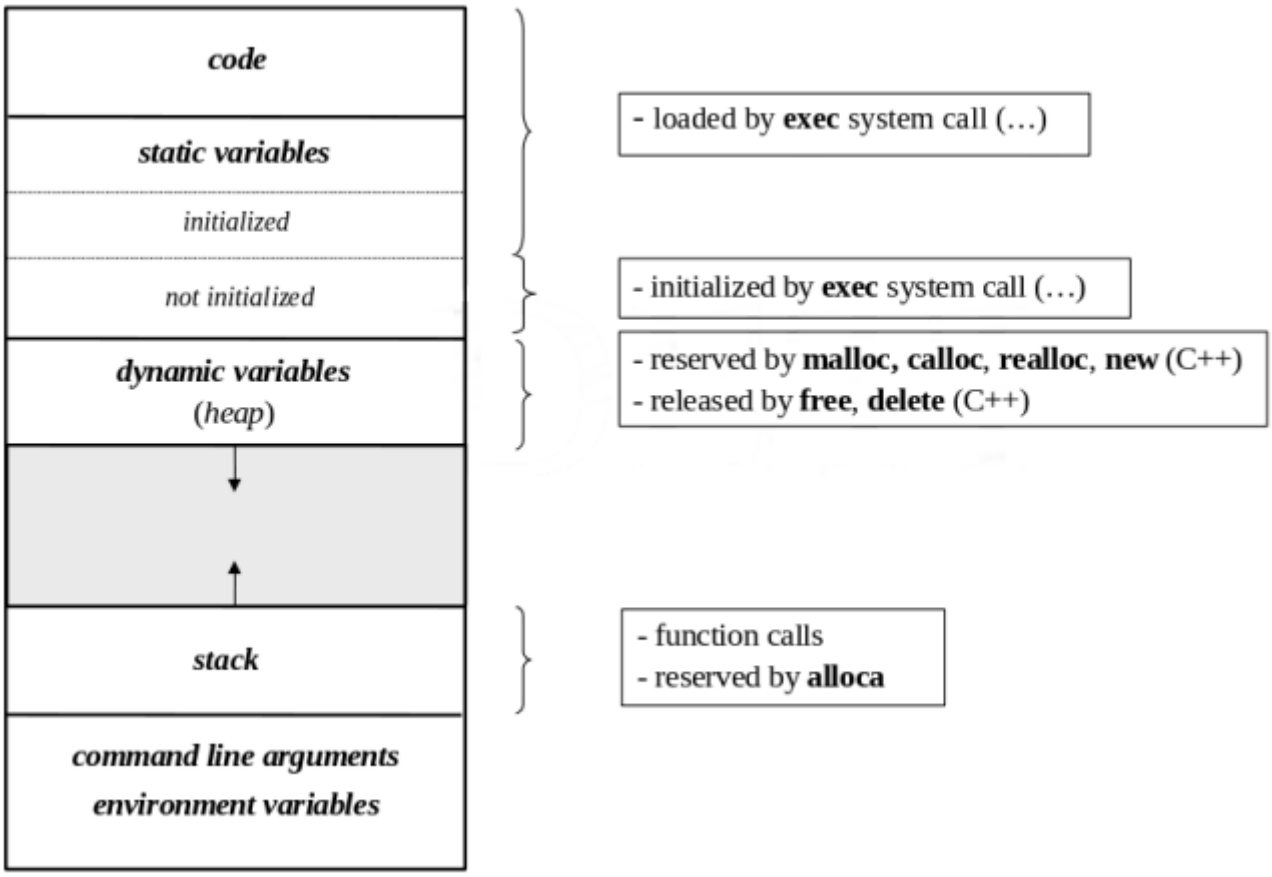
```
    /*normal work*/

    printf("helloworld1!\n");

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

    return 0;
}
```

Este código dá o seguinte output:

```
helloworld1!
atexit2
atexit1
```

Se em vez de um `exit_success`, termos um `exit_failure`, vai dar na mesma, porque a main, fazendo um return normal, acaba por chamar a `exit()`. Um exit_failure chama logo o `exit()`.

Address space de um processo



Introdução a Sistemas Operativos

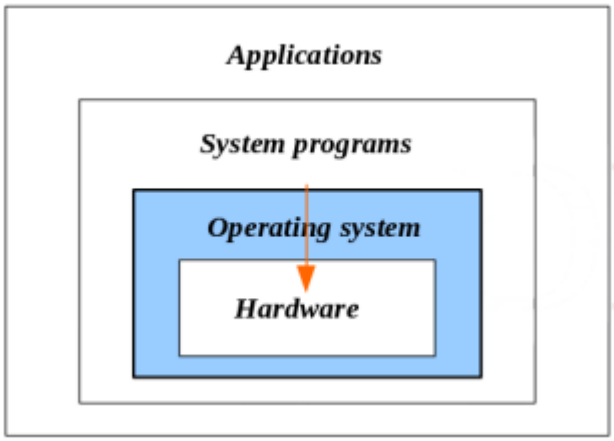
Perspetiva Global

Objetivos de um Sistema Operativo

- Dar vida ao hardware, fornecendo um ambiente de interação *abstrato*, agindo como uma interface entre a máquina e os programas (e.g chrome).
- Aloca dinamicamente recursos partilhados no sistema aos programas que estão a ser executados.
- Gere memória, processadores e outros aparelhos do sistema.
- Fornece **Conveniência** na forma que um pc é usado, **Eficiência** na utilização de recursos de um pc e **Habilidade para evoluir** para permitir o desenvolvimento de novas funções do sistema.

Existem duas perspetivas diferentes: *top-down* e *bottom-up*.

Sistema Operativo como uma máquina extended (outer-inner/user view)



O sistema operativo fornece uma vista abstrata do sistema underlying, abstraindo os programadores dos detalhes de hardware.
A interface com o hardware é um ambiente de programação uniforme, definido como um set de **system calls**, que permitem a portability de aplicações entre

sistemas computacionais estruturadamente diferentes. Isto é, podemos ter várias aplicações a correr no sistema operativo, porque definimos um **standard** de como as cenas devem correr. As aplicações podem ser mt diferentes, mas desde que obedeçam a esse standard de system calls, tudo bem.

Tipicas funcionalidades:

- Estabelecimento de ambiente de interação do user
- Fornecimento de equipamento para o desenvolvimento, testing e validação de programas.
- Fornecimento de mecanismos para a execução controlada de programas, incluindo intercomunicação e sincronização.
- Dissociation of the program's address space from the constraints imposed by the size of the main memory
- Controlar acesso ao sistema e a recursos especificos, protegendo contra acesso nao autorizado.
- Organização de memória secundária em file systems e controlar acesso a esses files.
- Definição de um modelo geral de acesso a I/O devices, regardless of their specific characteristics.
- Deteção de situaçoes de erro e gerar respostas apropriadas.

Sistema Operativo como um gerente de recursos (inner-outer/computational system view)

Um sistema computacional é um sistema composto de um set de recursos (processador(es), main memory, secondary memory, I/O device controllers) targeting the processing and storage of information.

O sistema operativo é o programa que gere o sistema computacional, fazendo a alocação controlada e ordenada dos seus diferentes recursos aos programas que competem por eles.

Resource usage is multiplexed in space and time

Alveja maximizar a performance do sistema computacional, assegurando o uso mais eficiente de recursos existentes.

Evolução de sistemas computacionais

Serial processing

- **Serial processing** - one user at a time.
- Utilizador tem acesso direto ao computador/processador.
- Não há sistema operativo.
- Problemas de scheduling:
  - Computador deve previamente ser reservado por utilizadores, possivelmente causando perdas de tempo ou saidas prematuras.
  - Uma quantia consideravel de tempo gasto a dar set up ao programa para correr.
- Software comum para todos os users.
- Utilização util de processador mt pobre.

Simple batch

Simple batch - one job at a time.

- Utilizador perde acesso direto ao processador.
  - Utilizador submete jobs em cartas ou tapes a um operador.
  - The operator joins jobs into a batch and places it in the computer input device
  - O monitor (batch OS) gere a execução dos jobs da batch no processador.
- Melhor que serial processing.
  - Scheduling feito pelo monitor, one job after the other.
  - Job control language helps improving setup time.
- Requisitos como proteção de memória e instruções priviligiadas aparecem.
  - (most of) the monitor and the user program must be in memory at the same time
- Utilização util de processador mt pobre.
  - Processador often at idle.

Multiprogrammed batch

Multiprogrammed batch - Enquanto um job está à espera da conclusão de uma operação I/O, outro job pode usar o processador. (**multiprogramming** or **multitasking).

- Scheduling, resource management and hardware support are required.

    - Monitor and all user programs (jobs) must be simultaneously in memory.
    - Interrupt and **DMA I/O** are needed to accomplish the I/O operation.
- Boa utilização do processador.
    - MAS, Não há interação direta do user com o computador

Time sharing

Time sharing - keeping different users, each one in a different terminal, in direct and simultaneous connection with the system.
- Quando computadores eram grandes e custavam muito, a unica opção era partilhá-los.

Diferenças com multiprogrammed batch:

- Tenta minimizar response time em vez de maximizar utilização de processador.
- Diretivas dadas ao OS por user commands em vez de job control language commands.

Em adição à proteção de memória e instruções priviligiadas, proteção de acesso a ficheiros e recursos (impressoras ...) são problemas a ser considerados.

Real-time operating systems

Objetivo - handle multiple processes at one time, ensuring that these processes respond to events within a predictable time limit.

Método - Variante de um sistema interativo que permite impor limites máximos a tempos de resposta de sistemas computacionais, dependendo do request.

Network operating systems

Objetivo - Aproveitar-se das interconnection facilites de computadores (a nivel de hardware) para estabelecer serviços comuns a uma comunidade inteira.

Services - File transfer (ftp); access to remote file system (NFS); resource sharing (printers, etc.); access to remote computers (telnet, ssh); e-mail; access to www (browsers).

Distributed operating systems

Objetivo - Aproveitar-se das facilities para construir multiprocessor computing systems, ou para interconectar sistemas computacionais diferentes, para estabelecer um ambiente de interação integrada onde o utilizador ve a computaçao paralela como uma entidade singular.

Metodologia - Assegurar completa transparência no acesso a processadores ou outros recuros do sistema distribuido, para que se permita:

- Static/dynamic load balancing.
- Aumentar speed de processamento, incorporando novos processadores ou novos sistemas.
- Paralelizamento de aplicações.
- Implementação de mecanismos de *fault tolerance*.

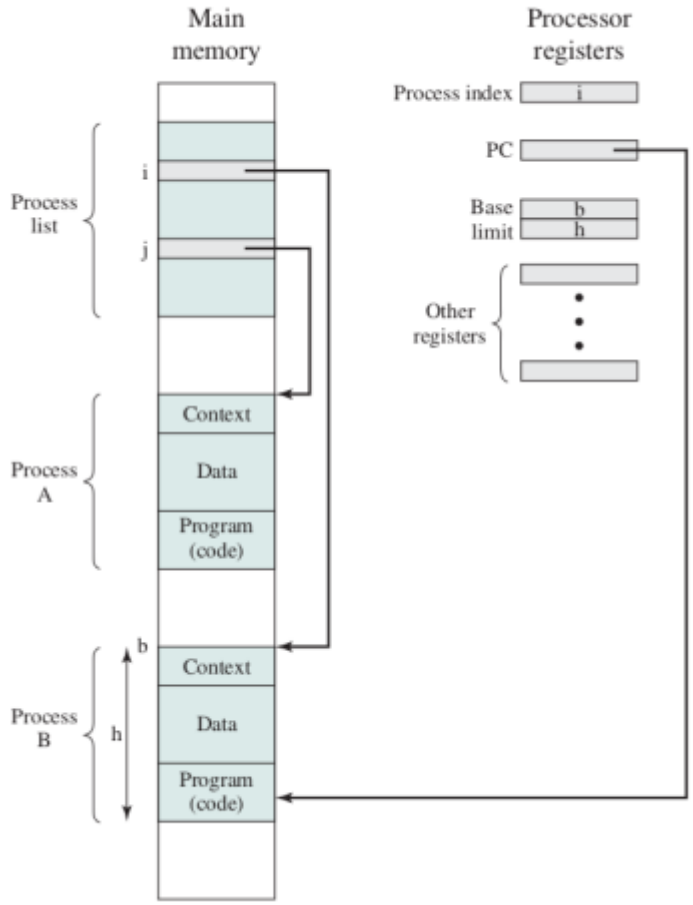
Tópicos chave

O processo

Aparece como um modo de controlo de atividade de varios programas executando num sistema multiprogramado, possivelmente pertencendo a diferentes users.

Cada processo ocupa um bloco de memória. Contém o programa, os dados e parte do contexto.
Cada processo ocupa um record numa lista de processos.

- Com um pointer para a memória do processo.
- E a outra parte do contexto.
  Um índice de processo identifica o processo a correr.
  Os registers de base e limite definem a memória do processo a correr.
  The program counter and all data references are checked against these register values, at every access.



Gerência de memória

Gerência de memória é necessário para suportar um uso flexivel de dados. Algumas funcionalidades:

- **Process isolation** - Processos independentes não podem interferir com a memória de outros processos.
- **Alocação e gerência automática** - Programas devem ser dinamicamente alocados na memória available.
- **Aumento dinamico de memória usada** - Memória usada por programas não deve ser definida no começo.
- **Controlo de acesso/proteção** - Partilha de memória deve ser possível, de forma controlada.
- **Armazenamento a longo termo** - Muitas apps precisam de armazenar informação por longos períodos de tempo, mesmo depois do pc encerrar.

Isto é alcançado com memória virtual e file systems.

- Memória virtual disassocia a memória vista por um processo e a memória real.

- Sistemas de ficheiros introduziram o conceito de ficheiro como o método de long-term storage.

Gerência e scheduling de recursos

Existe variedade de recursos partilhados entre processos - main memory, I/O devices, processors...

O sistema operativo deve gerir estes recursos e dar schedule ao seu uso pelos varios processos.
A alocação de recursos e policy de scheduling deve fornecer:

- **Fairness** - dar aproximadamente acesso igual e justo a recursos.
- **Responsiveness diferencial** - agir considerando o set total de requisitos.
- **Efficiency** - tentativa de maximizar throughput, minimizar response time, e, em caso de time sharing systems, acomodar o máximo numero de users. (Pode não ser possível, podem existir conflitos.)

Proteção e segurança de informação

Em time sharing systems, short-term e long-term data pode pertencer a diferentes utilizadores. Logo, proteção e segurança é um requisito, ou seja, controlar acesso a sistemas e a informação armazenada neles.

Pontos cobertos pelo OS:

- **Disponibilidade** - Proteger o sistema contra interrupção
- **Confidencialidade** - Assegurar que utilizadores não podem aceder a dados nao autorizados.
- **Integridade de dados** - Proteger dados de modificação nao autorixada.
- **Autenticidade** - Verificar a identidade de utilizadores e a validity of messages.

Semáforos e memória partilhada

Conceitos

Deadlock e starvation

Exclusão mútua no acesso a um recurso ou area partilhada pode resultar em:

- **Deadlock** - quando dois ou mais processos estão para sempre à espera para aceder à sua respetiva secção crítica, à espera por eventos que, demonstradamente, nunca irão acontecer. Operações ficam então bloqueadas.

- **Starvation** - quando dois ou mais processos competem por acesso a uma secção crítica e surgem novos processos que acedem primeiro. Operações são continuamente postponed.

Semáforos

Definição

Um semáforo é um mecanismo de sincronização, definido por um data type e duas operações atomicas, `up` e `down`.

Na sua estrutura de dados, possui um `unsigned int val` e uma queue de processos bloqueados.

Basicamente, temos duas operações, `down` e `up`. Digamos que um processo chama **`down`** num semaforo. Se o `val` tiver a 0, esse processo vai ser bloqueado (entra na queue de processos bloqueados). Se nao tiver a 0, decrementa `val` e procede.

Se chamar o **`up`**, vai incrementar o val, podendo acordar um processo bloqueado se houver (qd foi co `down`).

De notar que `val` pode apenas ser manipulado por estas operações e não é possivel checkar o valor de `val`.

É possivel implementar mutual exclusion atraves de semaforos binarios.

Bounded-buffer problem - problem statement

Neste problema, um número de entidades (produtores) produzem informação que é consumida por um número de entidades diferentes (consumidores).

Comunicação é feita por um buffer com bounded capacity (size fixo).

Assume-se que cada produtor e consumidor corre num diferente processo, então o FIFO deve ser implementado em **memória partilhada** para que os diferentes processos possam aceder a ele.

Como garantir que não surgem race conditions? Forçando mutual exclusion no acesso ao FIFO.

Bounded-buffer problem - Implementation

Portanto, um produtor teria que dar lock à FIFO para meter la cenas, e unlock depois de meter.

Então, teriamos um FIFO `fifo` declarado em *shared mem*, um semaforo `access` para controlar mutual exclusion, um semaforo `nslots` para controlar o numero de slots disponiveis (espaços vazios) e finalmente, um semaforo `nitems` para controlar o numero de items disponiveis (slots com cenas la dentro).

O produtor tem que fazer o seguinte, num loop infinito:

```
produce_data(&val);
sem_down(nslots); // meter o semaforo a vermelho para mais ninguem mexer nisto
sem_down(access); // meter o semaforo a vermelho, para ninguem aceder
fifo.insert(val); // produzir conteúdo
sem_up(access);   // meter a verde o acesso
sem_up(nitems);   // meter a verde o num de items.
do_something_else();
```

O consumidor tem que fazer o seguinte, num loop infinito:

```
sem_down(nitems); // meter a vermelho o num de items, vou mexer neles
sem_down(access); // meter o semaforo a vermelho, para ninguem aceder
fifo.retrieve(&val); // produzir conteúdo
sem_up(access);   // meter a verde o acesso
sem_up(nslots);   // meter a verde o num de slots.
consume_data(val);
do_something_else();
```

A solução errada seria no consumidor trocar a ordem aos dois `sem_down`. Isto possivelmente causaria deadlock, pq podia meter o `val` logo a 0, e nenhuma entidade conseguia dar up. Depende da sequencia de execuçao dos comandos, algo nao controlavel por nós nesta situaçao (race condition).

Análise de semáforos

Soluções concorrentes baseadas em semáforos têm vantagens e desavantagens.

Vantagens

- Suporte a nível de sistema operativo - Operações em semáforos sao implementadas pelo kernel e feitas disponiveis a programadores como system calls.
- Gerais - São construções low level e verstaeis, sendo capazes de ser usados em qualquer tipo de soluções.

Desavantagens

- Conhecimento especializado - O programador deve estar ciente de princípios de programação concorrente, uma vez que deadlock e race conditions facilmente podem acontecer.

Memória partilhada

Um recurso

- Address spaces de processos são independentes.
- Mas address spaces são virtuais.
- A mesma região física pode ser mapeada entre duas ou mais regiões virtuais.
- Memória partilhada é gerida como um recurso pelo sistema operativo.
- Duas ações são necessárias:
  - Requisitar um segmento de memória partilhada ao sistema operativo.
  - Mapear esse segmento na address space do processo.

Threads, mutexes e condition variables em Unix/Linux

Threads

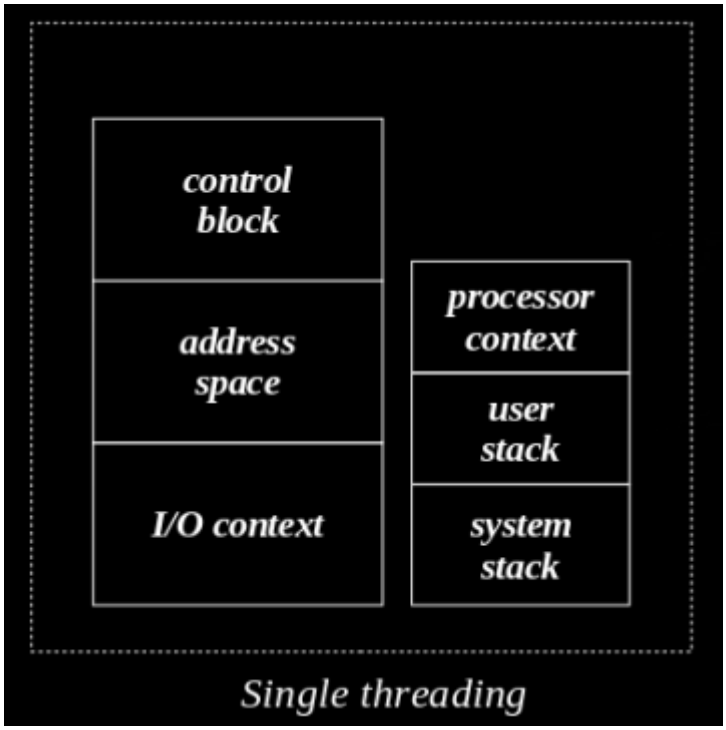
Single Threading

Em sistemas operativos tradicionais, um processo inclui:

- Address space (code and data of the associated program)

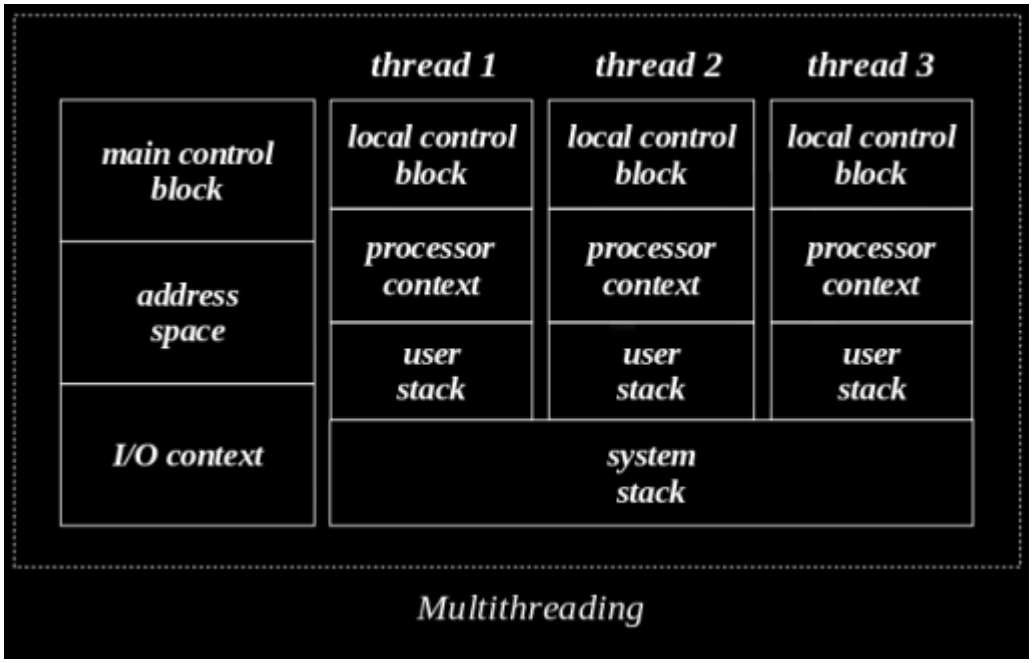
- Um conjunto de canais de comunicação com I/O devices
- Uma única thread de controlo, que incorpora os registers do processador (incluindo o program counter) e um stack.

Entretanto, estes componentes podem ser geridos separadamente.
Neste modelo, **thread** aparece como um componente de execução num processo.



Multithreading

Muitas threads independentes podem coexistir no mesmo processo, partilhando, assim, o mesmo address space e o mesmo contexto I/O. Isto é **multithreading**.
Threads podem ser vistas como **processos light weight**.

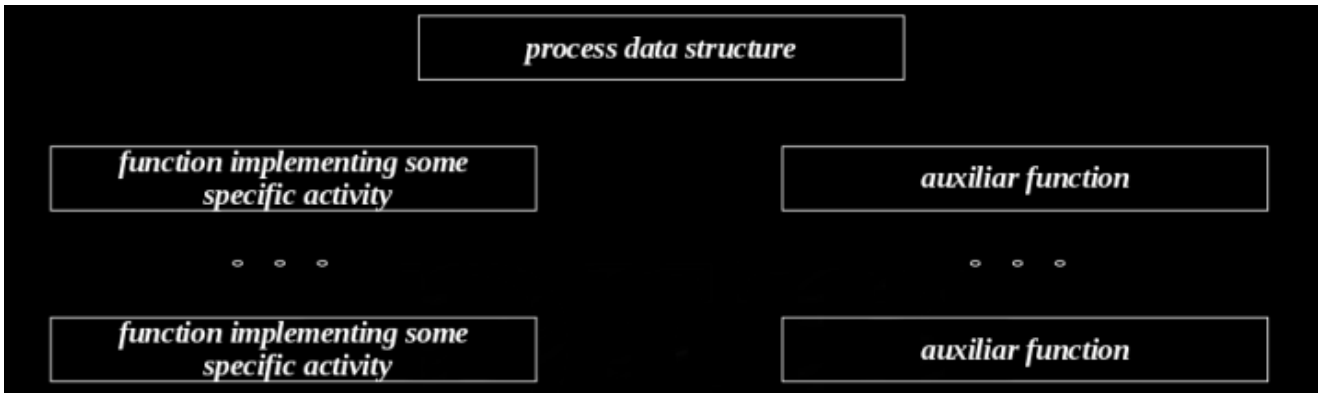


Structure of a multithreaded program

Cada thread é tipicamente associada à execução de uma função que implementa uma atividade específica.

Comunicação entre threads pode ser feita através do process data structure, que na perspetiva de threads, é global. Contém variaveis static e dynamic (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 ultima a ser destruida.



Implementações de multithreading

- **User level threads** - Threads são implementadas por uma biblioteca, a user level, que fornecem criação e gerência de threads se intervenção do kernel.
  - Versateis e portateis.
  - Quando uma thread chama uma blocking system call, o processo inteiro bloqueia, porque o kernel apenas ve os processos.
- **kernel level threads** - Threads são implementadas diretamente em kernel level.
  - Menos versateis e portateis.
  - Quando uma thread chama uma blocking system call, outra thread pode ser scheduled to execution.

Vantagens de multithreading

- **Implementação mais facil de aplicações** - Em muitas aplicações, decompor a solução num número de atividades paralelas faz o modelo de programação

mas simple. Uma vez que o address space e o contexto I/O são partilhados entre todas as threads, multithreading favorece esta decomposição.
- **Melhor gerência de recursos de computador** - Criar, destruir e trocar threads é mais facil do que fazer o mesmo com processos.
- **Melhor performance** - Quando uma aplicação involve I/O substancial, multithreading permite atividades fazerem *overlap*, acelerando assim a execução.
- **Multiprocessing** - Paralelismo real é possível se multiplos CPUs existirem.

Threads em Linux

A system call `clone`

Em Linux, há duas system calls para criar processo filho.

- `fork` - cria um novo processo que é uma full copy do atual.
  - I/O context e address space são duplicados.
  - O filho começa execução no ponto do forking.
- `clone` - cria um novo processo que pode partilhar elementos com o seu pai.
  - Address space, tabela de file descriptors e tabela de signal handlers sao partilhaveis.
  - O filho começa execução num função específica.

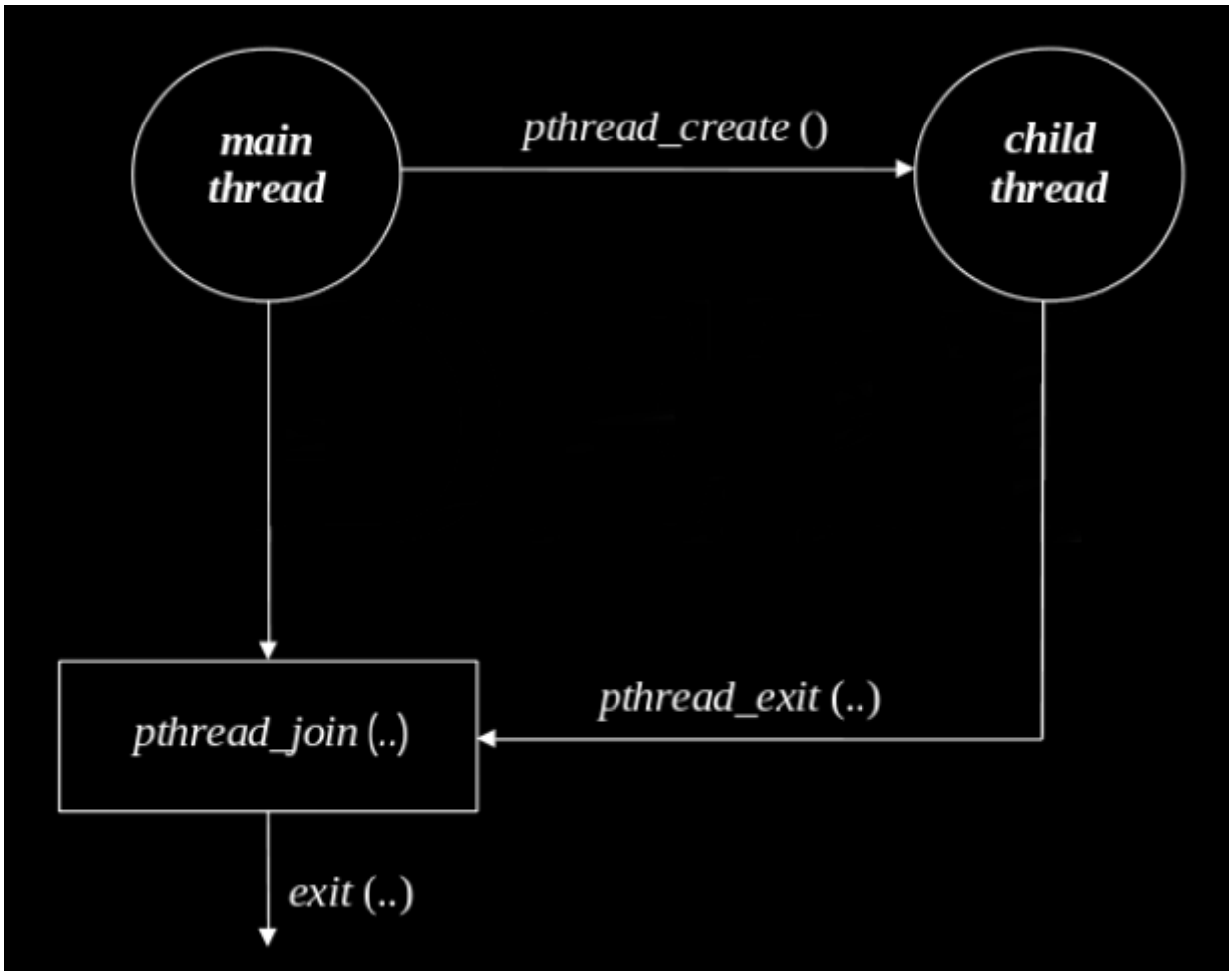
Assim, do ponto de vista do kernel, processos e threads são tratados de forma semelhante.

Threads do mesmo processo formam um thread group e têm o mesmo thread group identifier, *TGID* .

Dentro de um grupo, threads podem ser distinguidas pelo seu unique thread identifier, *TID*.

Criação e terminação de threads - biblioteca `pthread`

A thread main pode criar uma filha com a função `pthread_create()`. Depois a main dá `pthread_join()`, que a faz esperar pela terminação da thread filha. A thread filha faz `pthread_exit()`, encontrando-se no fim com a main no `pthread_join()`.



Exemplo

Função para a thread filha executar

```
int status;
void *func(void* par) {
        printf("Sou a thread filha a executar isto!\n");
        status = EXIT_SUCCESS;
        pthread_exit(&status);
}
```

Main

```
int main (int argc, char *argv[]*) {
        // lançar a thread filha
        pthread_t thr;
        pthread_create(&thr, NULL, threadChild, NULL)

        // se chegamos aqui, thread filha foi criada com sucesso
        // esperar por terminação da thread filha

        pthread_join(thr, NULL)
```

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

        return EXIT_SUCCESS;
}
```

Monitors

Introdução

Um problema com semáforos é que eles são usados para implementar **mutual exclusion *E*** para **sincronizar processos**.

Sendo primitivas low level, elas são aplicadas numa perspetiva *bottom-up*.

- Se condições necessárias nao forem satisfeitas, processos sao bloqueados antes de entrarem nas suas secções críticas.
- Esta approach é suscetivel a erros, principalmente em situaçoes complexas, porque pontos de sincronização podem estar espalhados pelo programa.

Uma approach higher level deve seguir uma perspetiva *top-down*.

- Processos devem primeiro entrar nas suas secções críticas e depois bloquear se condições de continuation não são satisfeitas.

Uma solução é introduzir uma construção (concorrente) no nivel de programação que lide com mutual exclusion e sincronização, **separadamente**.

Um monitor é um mecanismo de sincronização.
A biblioteca `pthread` fornece primitivas que permitem implementar monitors do tipo Lampson-Redell.

Definição

Uma aplicação é **vista como um conjunto de threads** que **competem para ter acesso** a estruturas de dados partilhadas. Estes dados partilhados podem apenas ser acedidos atraves de metodos de acesso. Cada metodo é **executado em mutual exclusion**.

Se uma thread chama um metodo de acesso enquanto outra thread está dentro de outro metodo de acesso, a sua execução é bloqueada até que a outra saia.

Sincronização entre threads é possível atraves de *condition variables*.

- Duas operações são possíveis:
  - `wait` - a thread é bloqueada e posta fora do monitor.
  - `signal` - se ha threads bloqueadas, uma é acordada. Mas qual?

Bounded-buffer problem - resolvido com monitors

Declaram-se 4 elementos como shared:

- O **FIFO** partilhado `fifo`.
- O **mutex** `access`, para controlar mutual exclusion.
- A variavel de condição **cond** `nslots`, para controlar disponibilidade de slots.
- A variavel de condição **cond** `nitems` para controlar disponibilidade de items.

Código do produtor num loop infinito:

```
produce_data();
lock(access);

if/while (fifo.isFull()) {
        wait(nslots, access);
}
fifo.insert(data);
signal(nitems);
unlock(access);
```

Código do consumidor num loop infinito:

```
lock(access);

if/while (fifo.isEmpty()) {
        wait(nitems, access);
}
fifo.retrieve(data);
signal(nslots);
unlock(access);
```

As secções críticas estão entre o `lock` e `unlock`.

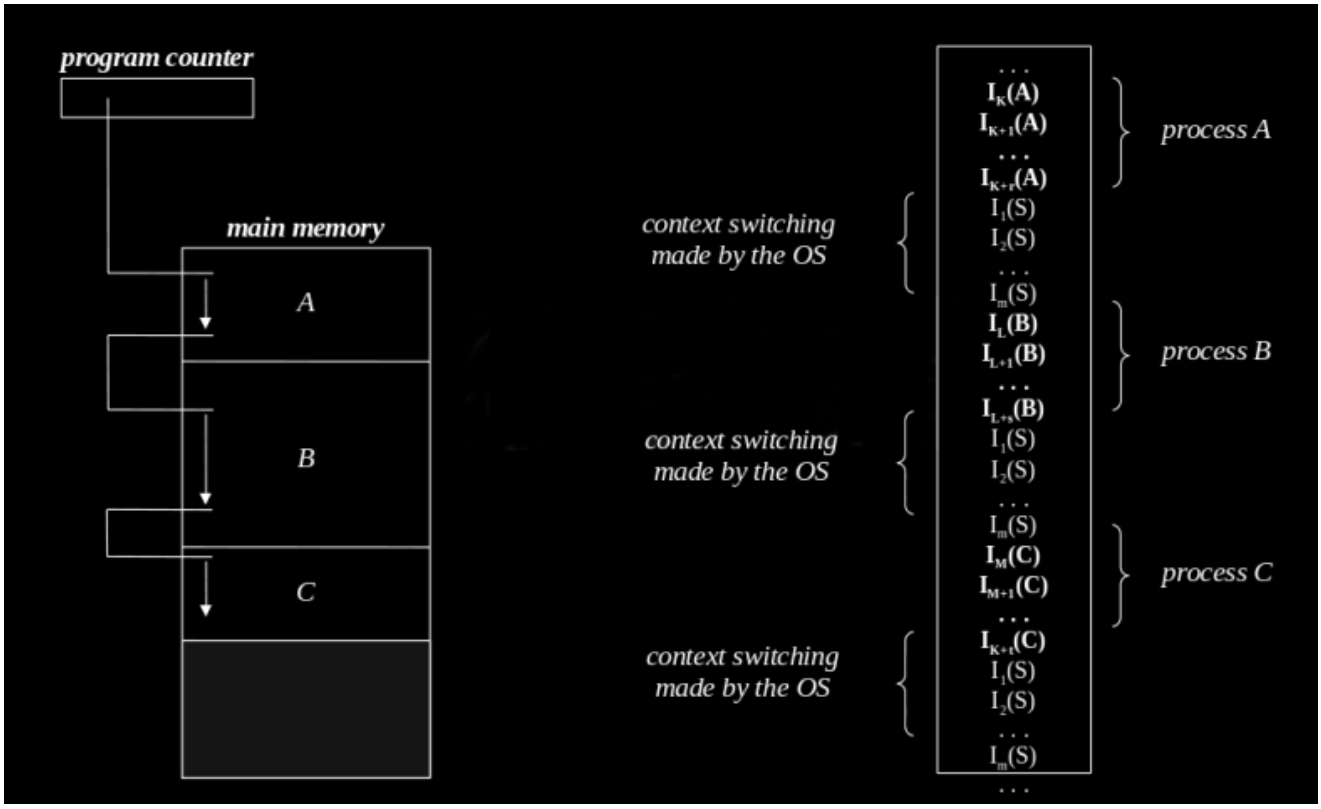
Usa-se a biblioteca `pthread`.
Permite implementação de monitors (do tipo Lampson/Redell) em C/C++.

Processos

Processo

Execução num ambiente multiprogrammed

Multiprogrammed é aquela cena de ilusão de conseguir correr varios programas simultaneamente.



Modelo de processo

Em multiprogramming, a atividade do processador é dificil de perceber, uma vez que ta sempre switching back and forth de processo a processo.

Então, é melhor assumir a existencia de um numero de processadores virtuais, um por processo existente.

- Desligando um processador virtual e ligando outro, corresponde a *process switching*.
- Número **real** de processadores >= Número **ativo** de processadores **virtuais**.

O switching entre processos, ou seja, entre processadores virtuais, pode ocorrer por diferentes razões, possivelmente nao controladas pelo programa a correr.

Assim, para ser viável, o modelo de processo requer que:

- A execução de qualquer processo não seja afetada pela localização no codigo, ou instância no tempo onde o switching acontece.
- Não hajam restrições impostas nas vezes totais ou parciais de execução de qualquer processo.

Estados de processo (short-term)

Um processo pode estar a não correr por diferentes razões. Devemos então identificar os possiveis estados.

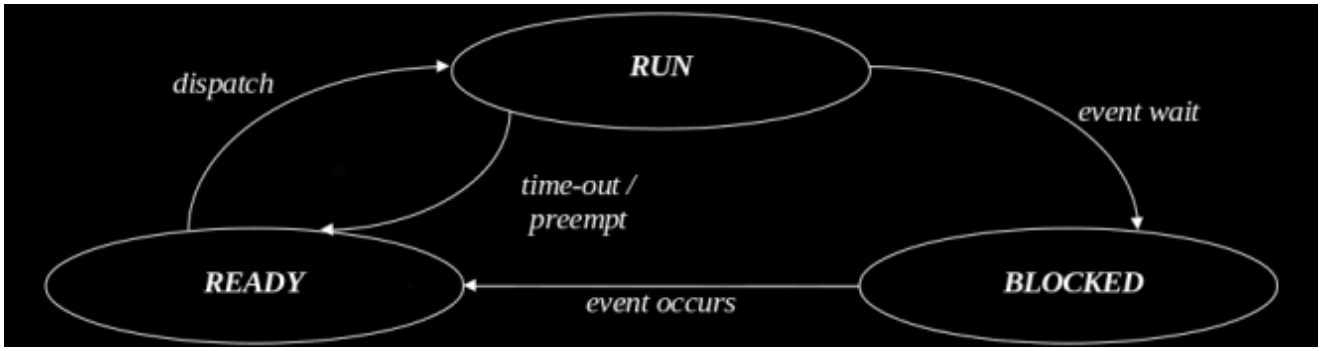
Mais importantes:

- **RUN** - O processo está na posse de um processador, a correr.
- **BLOCKED** - O processo está à espera da ocurrência de um evento externo (acesso a recurso, operação I/O...).
- **READY** - O processo está pronto a correr, mas à espera da disponibilidade de um processador para começar/continuar a sua execução.

Transições entre estados normalmente resultam de intervenção externa, mas nalguns casos, podem ser provocados pelo próprio processo.

A parte do sistema operativo que trata destas transições é o **(processor) scheduler**.

Diagrama



- **Event wait** - o processo a correr é bloqueado e fica à espera que um evento externo ocorra.
- **Dispatch** - um dos processos em **READY** é selecionado e dado ao processador.

- **Event occurs** - um evento externo ocorre e o processo em **BLOCKED** fica em **READY**.
- **Preempt** - um processo de maior prioridade fica **READY**, entao o processo em **RUN** é removido do processador.
- **Time-out** - o tempo atribuido ao processo acaba, entao o processo é removido do processador.

Estados de processo (medium-term)

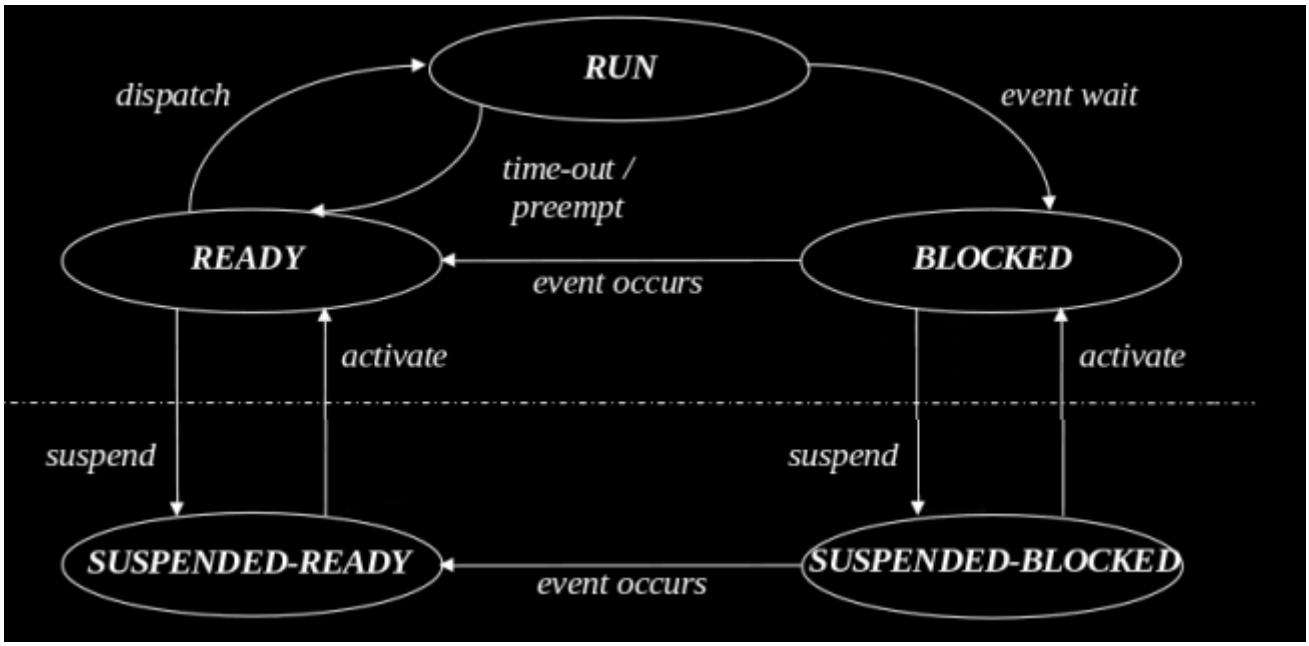
A memória principal é finita, o que limita o número de processos coexistentes.

Uma forma de superar esta limitação é usar uma área em memória secundária para extender a memória principal. Esta área chama-se *swap area*, um processo non-running, ou parte, pode ser **swapped out**, para que se liberte memória principal para outros processos. Esse processo é mais tarde **swapped in**, assim que memória principal fique disponivel.

Dois novos estados podem ser adicionados ao diagrama para incorporar estas situações:

- **suspended-ready** - o processo está ready mas swapped out.
- **suspended-blocked** - o processo está bloqueado e swapped out.

Diagram



Estados e transições de processo (long-term)

O diagrama prévio assume que processos são intemporais. Isto não é verdade. Processos são criados, existem durante algum tempo e eventualmente terminam.

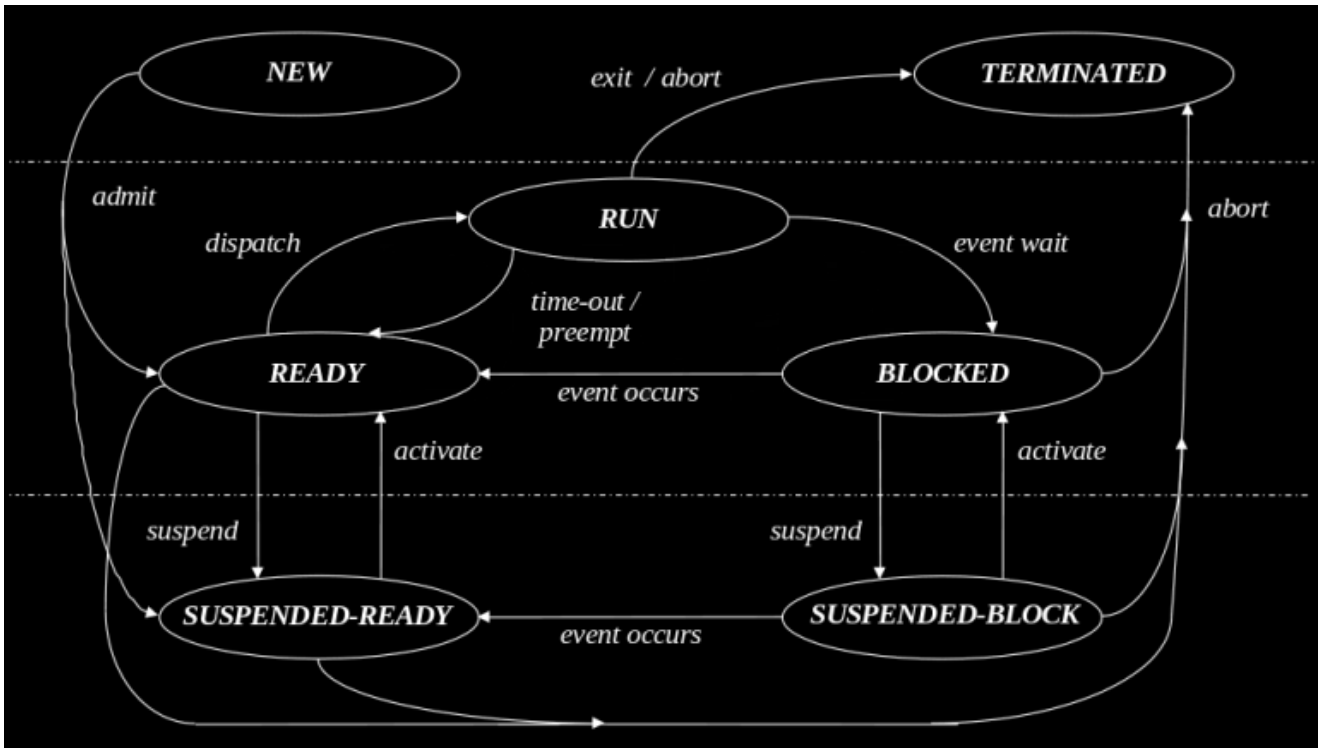
Dois novos estados são necessários para demonstrar criação e terminação:

- **new** - o processo foi criado mas ainda nao foi admitido à pool de processos executaveis.
- **terminated** - o processo foi libertado da pool de processos executaveis, mas algumas açoes ainda sao necessarias antes do processo levar discard.

Três novas transições existem:

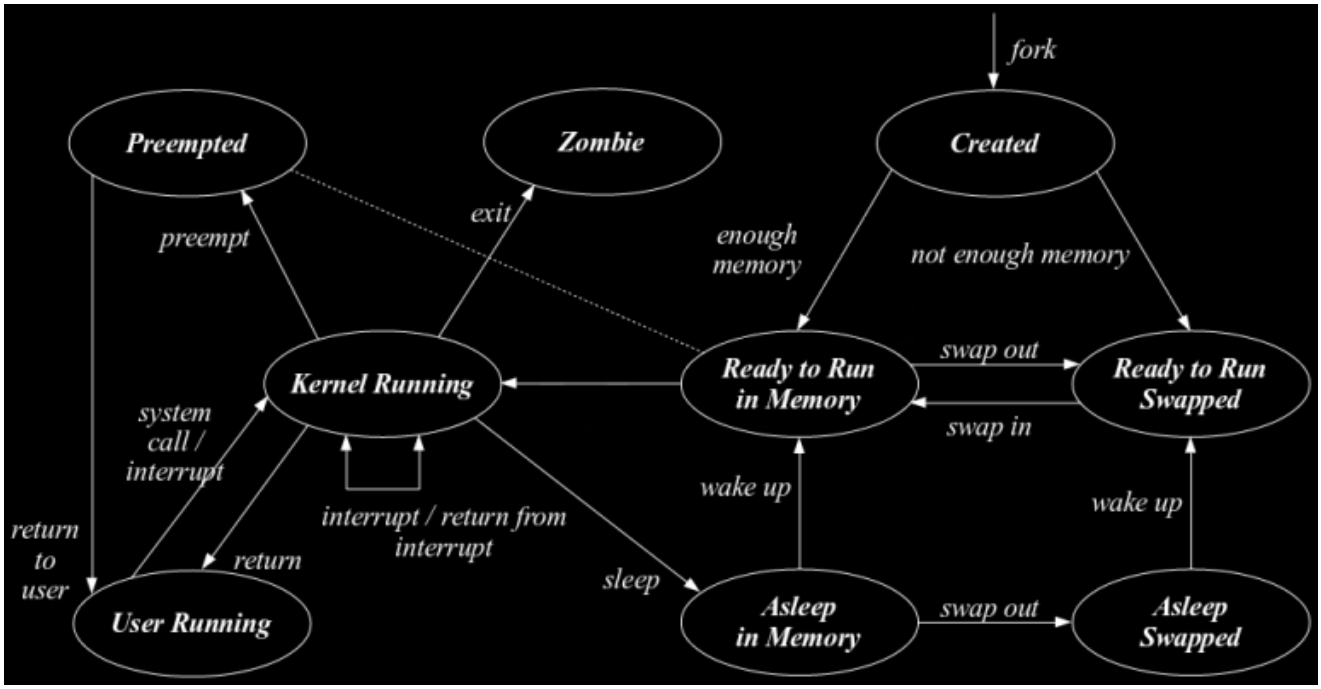
- **admit** - o processo é aceitado (pelo OS) à pool de processos executaveis.
- **exit** - o processo a correr indica ao OS que acabou
- **abort** - o processo é forçado a terminar (por causa de erro fatal ou um authorized process aborta a sua execução).

Diagrama global de estados



Um processo **NEW** apenas pode levar *admit*. Isto leva-o a **READY** ou, no caso de memória principal insuficiente, **SUSPENDED-READY**.
Em todos os estados menos **NEW** se pode levar um *abort*, que leva a **TERMINATED**.
O *exit* acontece apenas no **RUN**, que leva a **TERMINATED**.

Diagrama típico de estados em Unix



Há dois estados **run** associados ao running mode do processador, *kernel running* (supervisor) e *user running* (user).

Quando um user process sai do modo supervisor, pode levar **preempt**, porque um processo de prioridade maior pode ficar ready to run.

Process control table

Para implementar o process model, o sistema operativo precisa de uma data structure usada para armazenar informação sobre cada processo - **process control block**.

A **process control table (PCT)** armazena process control blocks.

Context switching

Processadores atuais têm dois modos de funcionamento:

- **Supervisor mode** - todo o instruction set pode ser executado. **Modo priviligiado**.
- **User mode** - apenas parte do instruction set pode ser executado. **Modo normal de operação**. instruções I/O sao excluidas assim como as que modificam control registers.

Trocar de user para supervisor mode é apenas possivel atraves de uma **Exception** (por razoes de segurança).

Uma exceção pode ser causada por:

- **Interrupção I/O** - externa à execução da instrução atual.
- **Instrução ilegal** (div por 0, bus error) - associada à execução da instrução atual, mas a exception não é feita de proposito.
- **Trap instruction** (software interruption) - associada a execução da instrução atual, e feita de propósito.

O sistema operativo deve funcionar em supervisor mode, para que tenha acesso a todas as funcionalidades do processador.

Assim, funçoes do kernel devem ser feitas por hardware (interrupt), ou trap (sw interrupt).

Isto estabelece um ambiente de operação uniforme: **handling de exceptions**.

Context switching é o processo de armazenar o estado de um processo e restorar o estado de outro. Ocorre necessariamente no contexto de uma exception, com uma pequena diferença de como é handled.

Processando uma exceção (normal)

1. Save PC and PSW (of the executing process in the system stack)
2. PC := Exception handling routine (ou seja, vamos para a rotina de exception handling)
3. Save context
4. Processing
5. Restore context
6. Restore PC and PSW (...)

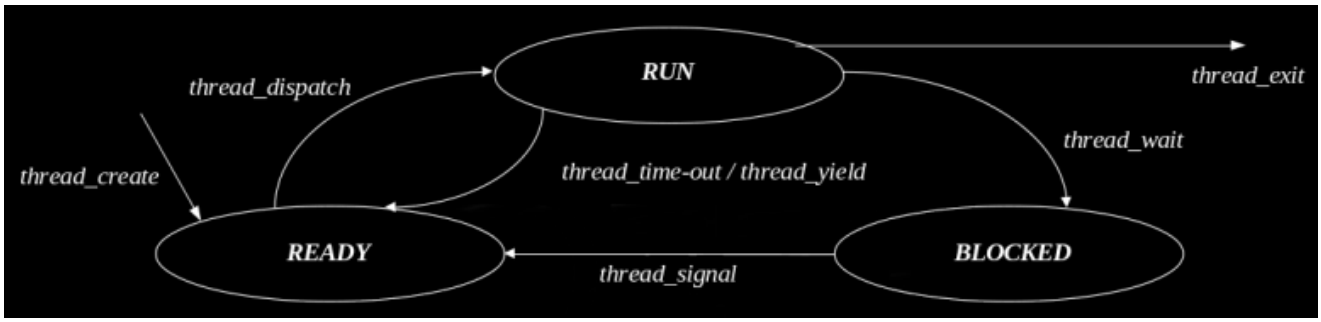
Processando um process switching

1. Save data of running process (address space, registers, I/O data, ...) in *PCT*
2. Move process to the appropriate queue and update process state (also in *PCT*). Ou seja, fazer ou time-out ou wait event ou exit/abort.
3. Select a process in the **READY** state (according to some policy)
4. Update state of selected process (in *PCT*)

5. Restore date of selected process (...)

Threads

Diagrama de estados de uma thread



Apenas estados sobre a gerencia do processador sao considerados. (short-term states).
Os estados *suspended-ready* e *suspended-blocked* nao estao presentes porque estao relacionados ao processo e nao a threads.
Os estados *new* e *terminated* tambem nao estao presentes. The management of the multiprogramming environment is basically related to restrict the number of threads that can exist within a process.

Processor scheduling

Processor scheduler

Definição

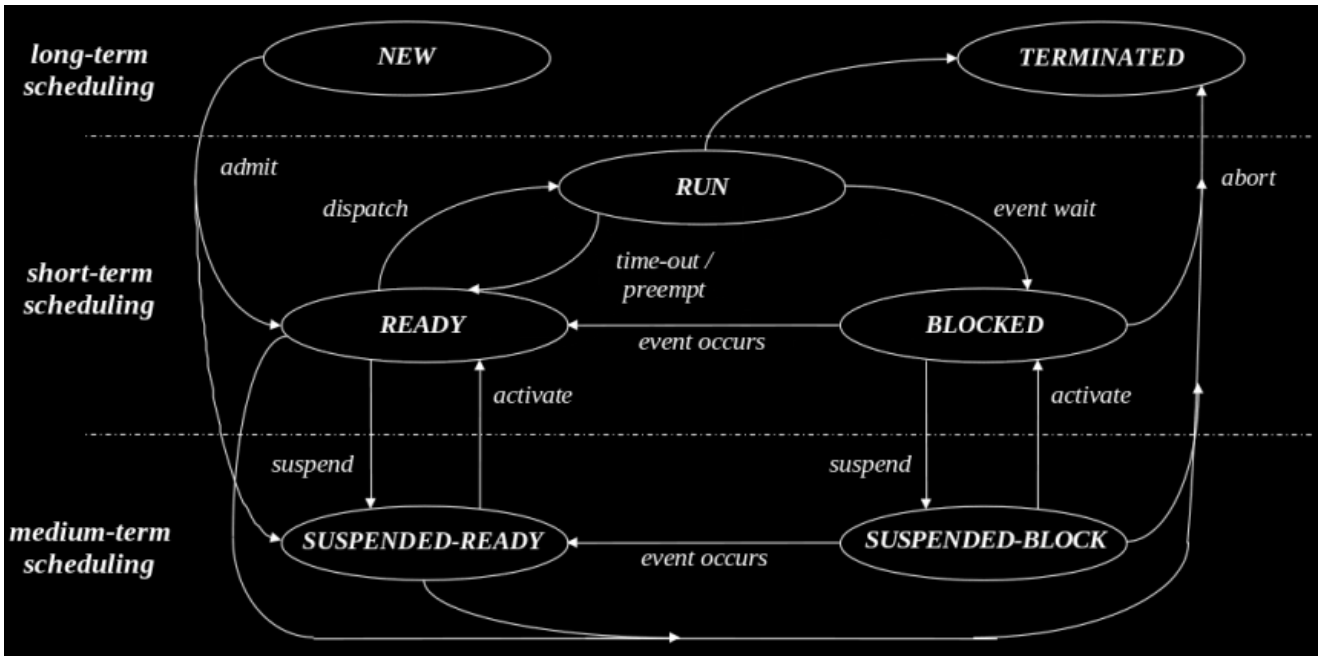
A execução de um processo é uma sequência alternada de dois tipos de periodos:

- **CPU burst** - Executar instruções de processador.
- **I/O burst** - Esperar pelo fim de um I/O request.

Então, um processo pode ser **I/O-bound** (pede muitos I/O requests, logo ha muitos CPU bursts de pouca duração), ou **CPU-bound** (o contrario).

A ideia atras de multiprogramming é **aproveitar-se dos periodos de I/O burst para meter outros processos a usar o CPU**.

Niveis em processor scheduling



Long-term scheduling

- Determina quais programas sao aceitados para serem processados.
- Controla o grau de multiprogramming
- Se aceite, um job ou um user program torna-se um processo e é adicionado à queue de processos **ready**.

Medium-term scheduling

- Gere a swapping area
- Decisao de swap in é baseada no grau de multiprogramming
- Gerencia de memoria tambem pode condicionar swapping in.

Short-term scheduling

- Decide qual o proximo processo a ser executado.
- É invocado sempre que um evento ocorre e bloqueia o processo atual ou permite o preempt.
- Eventos possiveis: clock interrupts, I/O interrupt, system call, signal (de semaforo).

Short-term processor scheduler

Preemption & non-preemption

O short-term processor scheduler pode ser preemptive ou non-preemptive.

Non-preemptive scheduling - Um processo mantem o processador ate se bloquear ou acabar.

- Transições time-out e preempt nao existem.
- Tipicos em sistemas batch

Preemptive scheduling - Um processo pode perder o processador devido a razoes externas.

- Por dar time-out, ou por preempt.

Sistemas interativos têm quer ser preemptive.

Scheduling algorithms

Evaluation criteria

O objetivo principal de short-term scheduling é alocar tempo de processador para otimizar alguma função objetiva do comportamento do sistema.

Tem que se utilizar criterios.

Estes criterios podem ser vistos de perspetivas diferentes:

- **User-oriented** - relacionado ao comportamento do sistema no ponto de vista do user individual ou processo.
- **System-oriented** - relacionado à utilizaçao eficiente do processador.

Os criterios sao interdependentes, sendo assim impossivel otimizá-los todos simultaneamente.

User-oriented scheduling criteria

Turnaround time - intervalo de tempo entre a submissao de um processo/job e o seu fim. (inclui tempo de execução, tempo de espera por recursos(cpu incluido)). Adequado a batch job. Deve ser minizado.

Waiting time - soma de periodos gastos num processo à espera do ready state. Deve ser minimizado.

Response time - tempo desde o pedido de submissao atè à resposta.

Deadlines - tempo de conclusao de um processo.

Predictability - como a resposta é afetada por load no sistema. Um job deve correr à volta do tempo/custo esperado regardless do load no sistema.

System-oriented scheduling criteria

Fairness - Igualdade de tratamento. Na ausencia de guia, processos devem ser tratados da mesma forma. nenhum processo deve sofrer starvation.

Throughput - numero de processos completados por unidade de tempo. Mede a quantia de work feita pelo sistema. Depende da average length de processos mas tambem na scheduling policy.

Processor utilization - percentagem de tempo que o processador ta busy. Deve ser maximizado.

Enforcing priorities - Processos de maior prioridade devem ser favorecidos.

É impossivel satisfazer todos os criterios ao mesmo tempo. Depende da aplicação para decidir quais satisfazer.

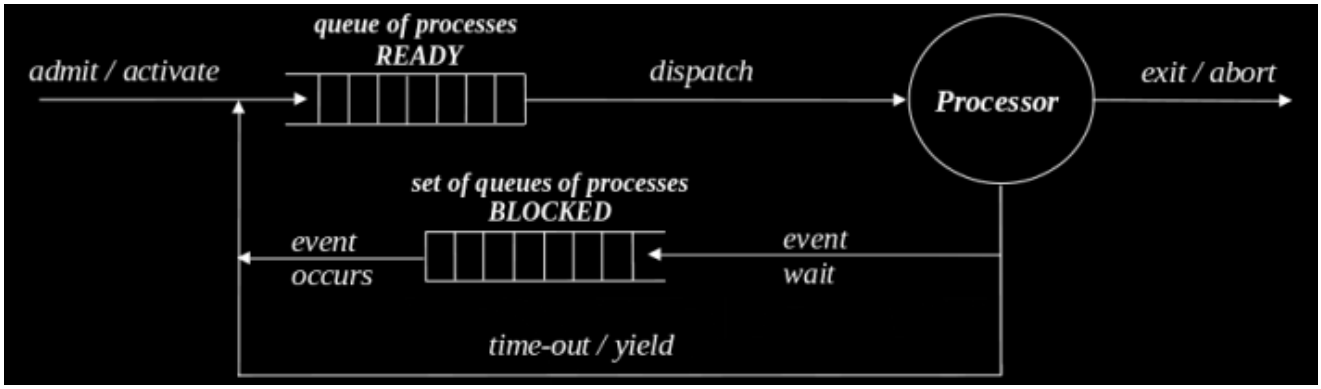
Prioridades

Sem prioridades

Todos os processos sao iguais, sendo servidos em ordem de chegada à queue **READY**.

- Em scheduling non-preemptive, é normalmente referido como **FCFS** - First Come First Serve. Nao existe transiçao time out.
- Em scheduling preemptive, é normalmente referido como **round robin**. Existe time out.

Facil implementar, favorece processos cpu-bound.



Forçando prioridades

Muitas vezes, todos os processos tratados da mesma forma nao é adequado.

- Em sistemas interativos, minimizar response time requer que processos I/O bound sejam priviligiados.

Para tratar disto, processos devem ser agrupados em diferentes niveis de prioridade.

- Processos de maior prioridade sao tratados primeiro.
- Processos de menor prioridade podem sofrer **starvation**.

Tipos de prioridades

Prioridades podem ser:

- **static** - se nao mudam over time
- **deterministic** - se sao definidos deterministically.
- **dynamic** - se dependem no historico de execução dos processos.

Prioridades **estaticas**:

- Processos sao agrupados em classes de prioridades fixas, de acordo com a sua importancia relativa.
- **Risco de starvation** de lower-priority processes.
- A disciplina mais injusta.
- **Tipico em real time systems**.

Prioridades **deterministicas**:

- Quando um processo é criado, uma prioridade é lhe dado.
- Em time-out, a prioridade é decrementada
- Em even wait, é incrementada.
- Quando se chega ao valor minimo, a prioridade mete-se no valor inicial.

Prioridades dinamicas:
Falta materia daqui. Slides de processor scheduling. slide 19/34

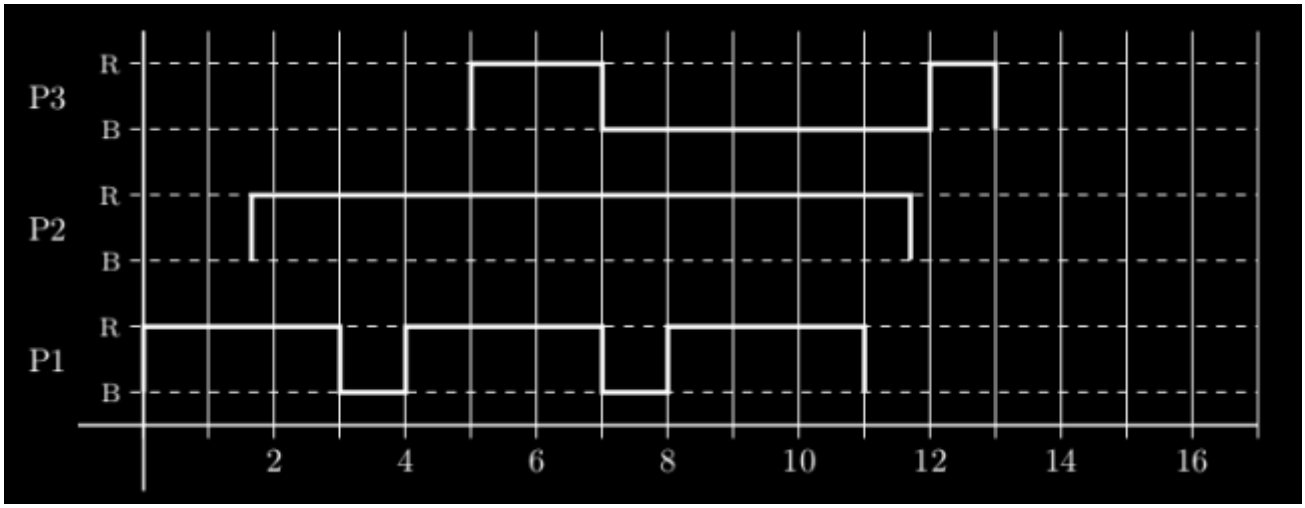
Scheduling policies

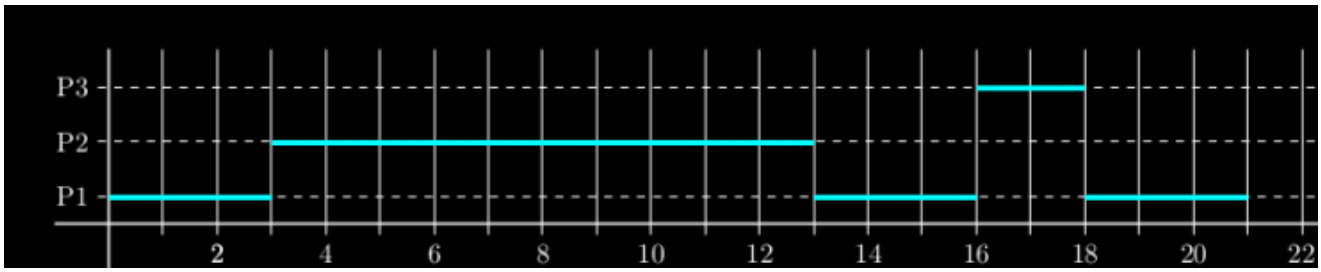
FCFS

First-Come-First-Server scheduler

- FIFO
- Non-preemptive, a *nao ser que combinado com um priority schema*.
- **Favorece processos CPU-bound**.
- Pode resultar no mau uso do cpu e dispositivos I/O.

Exemplo

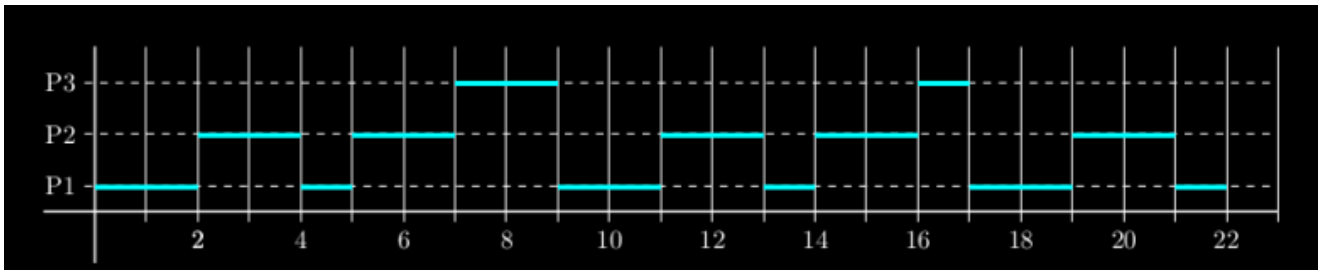




Round Robin (RR)

- Preemptive, cada processo tem time quantum antes de levar preempt. - **time slicing**
- O processo mais velho na ready queue é o primeiro a ser selecionado.
- Time quantum muito curto é bom porque processos curtos vao se mover rapidamente pelo sistema e response time é minimizada, mas é mau porque cada context switching involve **overhead**.
- Eficiente em sistemas general purpose time-sharing e de transações.
- Favorece processos CPU-bound.
- Pode resultar em mau uso de I/O devices.

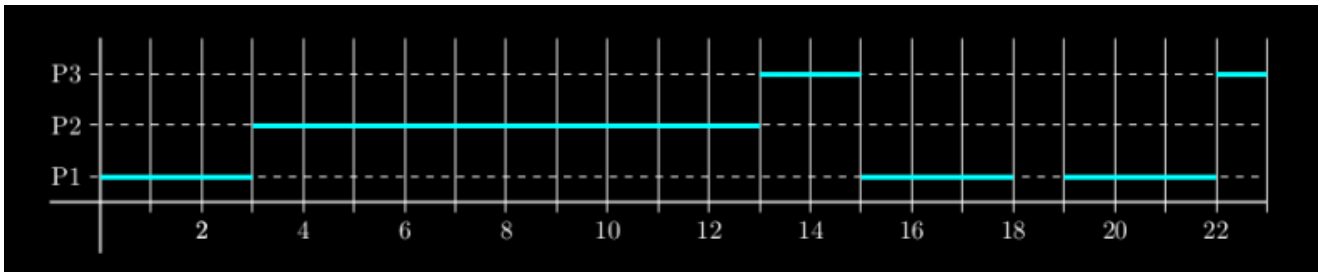
Exemplo assumindo time quantum de 2, sem prioridades (Mesmo enunciado do exemplo de FCFS)



Shortest Process Next (SPN)

- Non preemptive
- O processo com menor CPU burst time esperado é o proximo a ser selecionado. Usa-se FCFS para desempates.
- Throughput maximo.
- Minimo waiting time e turnaround time.
- Risco de starvation em processos longos.
- Requer conhecimento em advance do **expected** processing time.
  - Valor pode ser previsto, usando valores previos.
- Usado em long-term scheduling batch systems.

Exemplo (sem prioridades e assumindo o enunciado previo)



Politicas de Scheduling em Linux

Classes diferentes

- **SCHED_FIFO** - Fifo real-time threads, com prioridades e uma thread pode voluntariamente sair do processador
- **SCHED_RR** - Round-robin real-time threads, com prioridades.
- **SCHED_OTHER** - non-real-time threads.

Algoritmo para SCHED_OTHER

Completely fair scheduler (CFS) - baseado em virtual run time value, que ve quanto tempo uma thread ta a correr ate ao momento. O virtual run time está relacionado ao physical run time e à prioridade. As threads com maior prioridade levam shrink no virtual run time.

A thread com menor virtual run time é selecionada.
Uma thread de maior prioridade que fique ready pode dar preempt a outra.

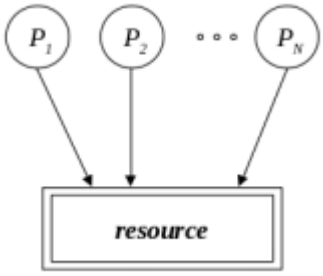
Comunicação interprocess

Conceitos

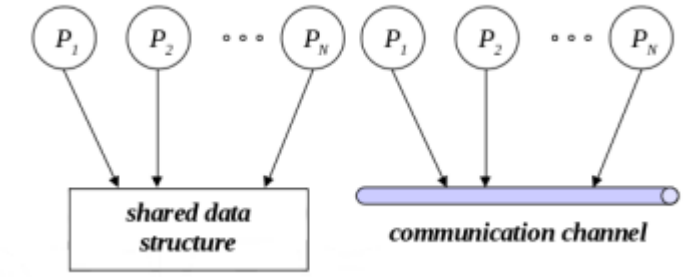
Processos independentes e colaborativos

Num ambiente multiprogrammed, dois os mais processos podem ser:

- **Independentes** - se eles, desde a sua criação ao seu fim, nunca explicitamente interagem.
  - Actually, há um interação implicita, pq competem por recursos do sistema.
  - ex: jobs num sistema batch, processos de users diferentes.
- **Cooperativos** - Se partilham informação ou comunicam explicitamente
  - A partilha requer um **address space comum**.
  - Comunicação pode ser feita através do address space comum ou um canal de comunicação que os conecte.



Processos independentes competindo por um recurso.
É a responsabilidade do sistema operativo assegurar a atribuição de recursos a processos, de forma a que não haja perda de informação.
Em geral, isto impoe que **apenas um processo** use o recurso **at a time** - **mutual exclusive access**.



Processos cooperativos partilhando informação ou comunicando.
É a responsabilidade dos processos assegurar que o acesso da area partilhada é feito de forma a que nao haja perda de informação.
Em geral, isto impoe que apenas um processo possa aceder à area partilhada **at a time** - **mutual exclusive access**.

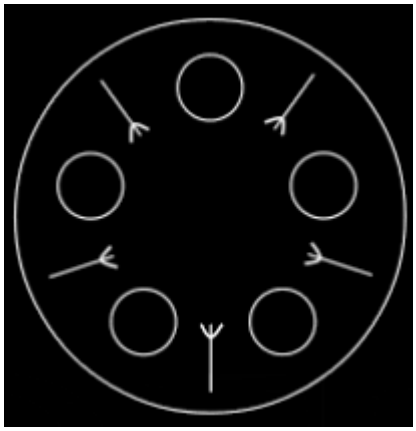
Critical section

Ter acesso a um recurso ou a uma area partilhada actually significa executar o código que faz o acesso.
Esta secção de código chamada **critical section**, se nao tiver protegida como deve ser, pode resultar em **race conditions**.
Uma **race condition** é uma condição onde o comportamento (output, resultado) **depende da sequência ou timing de eventos (incontrolaveis).**

Secções criticas devem então executar em **exclusão mutua**.

Philosopher dinner

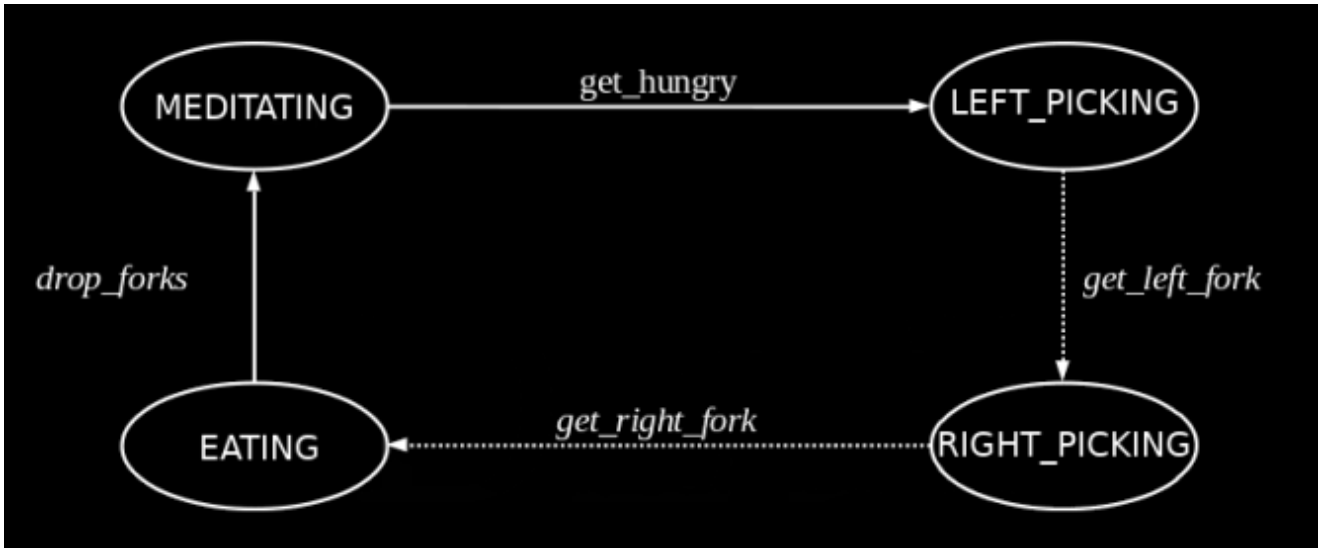
Problema



5 filosofos estão sentados numa mesa circular, com comida à frente deles.
Para comer, cada filosofo precisa de 2 garfos. À esquerda e à direita dos prato há um garfo.

Eles ficam algum tempo sem usar os garfos para mastigar a comida.
Digamos que cada filosofo é um processo ou thread e os garfos sao recursos.

Solução



Esta solução sofre de race conditions.

Mutual exclusion a um recurso pode resultar em **deadlock**.
Pode tambem resultar em **starvation**.

Primitivas de acesso

Requisitos

Requisitos que devem ser observados numa secção crítica:

- **Mutual exclusion efetiva** - Acesso à secção crítica associada ao mesmo recurso deve ser apenas permitido a um processo at a time.
- **Independence** on the number of intervening processes or on their relative speed of execution
- Um processo fora da sua secção critica nao pode prevenir outro processo de entrar na sua secção critica.
- **No starvation** - um processo que pede acesso à sua secção critica nao deve ter que esperar para sempre.
- **Length** de uma critical section tem que ser **finita**.

Tipos de soluções

Em geral, uma **memory location** é usada para controlar acesso a secções críticas. Funciona como uma *binary flag*.

Dois tipos de soluções:

- **Software solutions** - Soluções que sao baseadas em instruções tipicas, usadas para aceder à memory location.
  - Read e Write feitos por diferentes instruções.
  - Interrupções podem acontecer entre reads e writes.
- **Hardware solutions** - soluções baseadas em instruções especiais para aceder à memory location.
  - estas instruções permitem o read e depois escrever a memory location numa forma atomica.

Solução de software

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;
}
```

idk

Solução de hardware

Desativando interrupts

Sistema computacional de apenas um processador

- Process switching, num ambiente multiprogrammed, é sempre causado por um aparelho externo:
  - **Real time clock (RTC)** - Causa a transição **time-out** em sistemas preemptive.
  - **Device controller** - pode causar transições preempt no caso de um processo com maior prioridade ficar **READY**.
  - Em qualquer caso, interrupções do processador.

Assim, acesso à mutual exclusion pode ser implementada desativando interrupts. Apenas valido em kernel, porque codigo buggy ou malicioso pode bloquear o sistema.

Sistema computacional de multiplos processadores

Desativar interrupts num processador nao tem qualquer efeito.

Instruções especiais - Test and Set (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;
}
```

Instruções especiais - Compare and Switch (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;
}
```

Busy waiting

Estas 2 soluções sofrem de *busy waiting*. A primitiva `lock` está a usar o CPU enquanto espera. Chama-se *spinlock*, porque o processo gira à volta da variavel enquanto fica à espera de acesso.

Em sistemas uniprocessor, nao queremos busywaiting, por haver perda de eficiencia e risco de deadlock.

Em sistemas multiprocessor, busy waiting pode ser menos perigoso.

Block and wake up

Em geral, pelo menos em sistemas de uniprocessor, é preciso bloquear o processo enquanto tá à espera de entrar na sua secção critica.

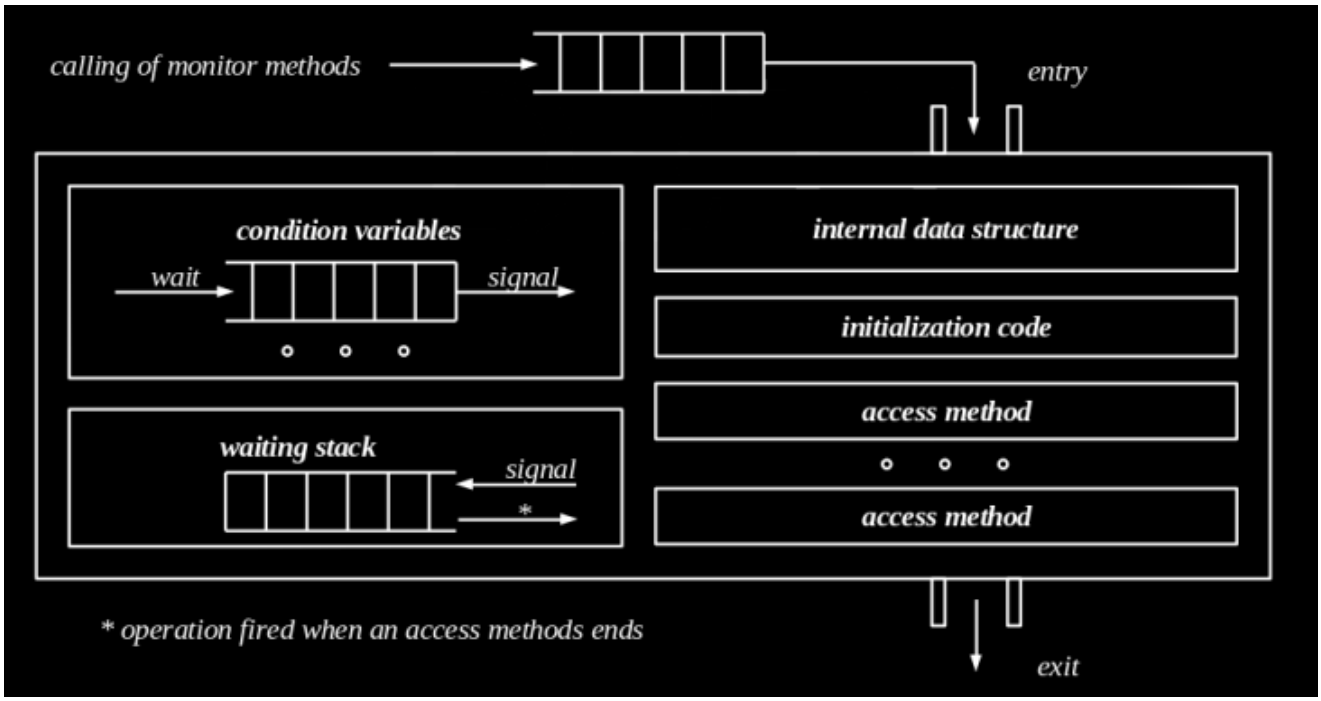
Monitors

Lembrar da definição de monitors - mecanismo de sincronização. usa threads, condition variables e mutex. wait, signal, etc...

Hoare

O que fazer quando ocorre o `signal`?

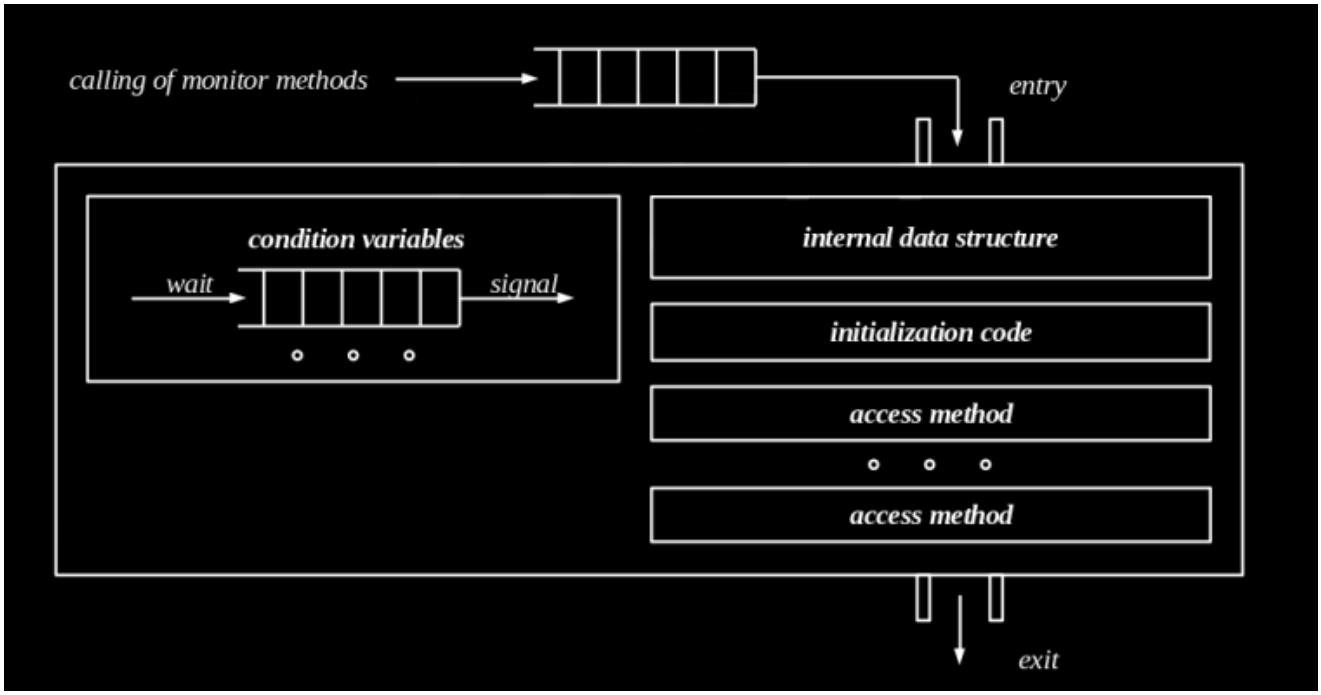
Hoare monitor - a thread que chama o `signal` é posta fora do monitor, para que a thread acabada de acordar possa proceder. Requer stack, onde a thread bloqueada é posta.



Brinch Hansen

O que fazer quando ocorre o `signal`?

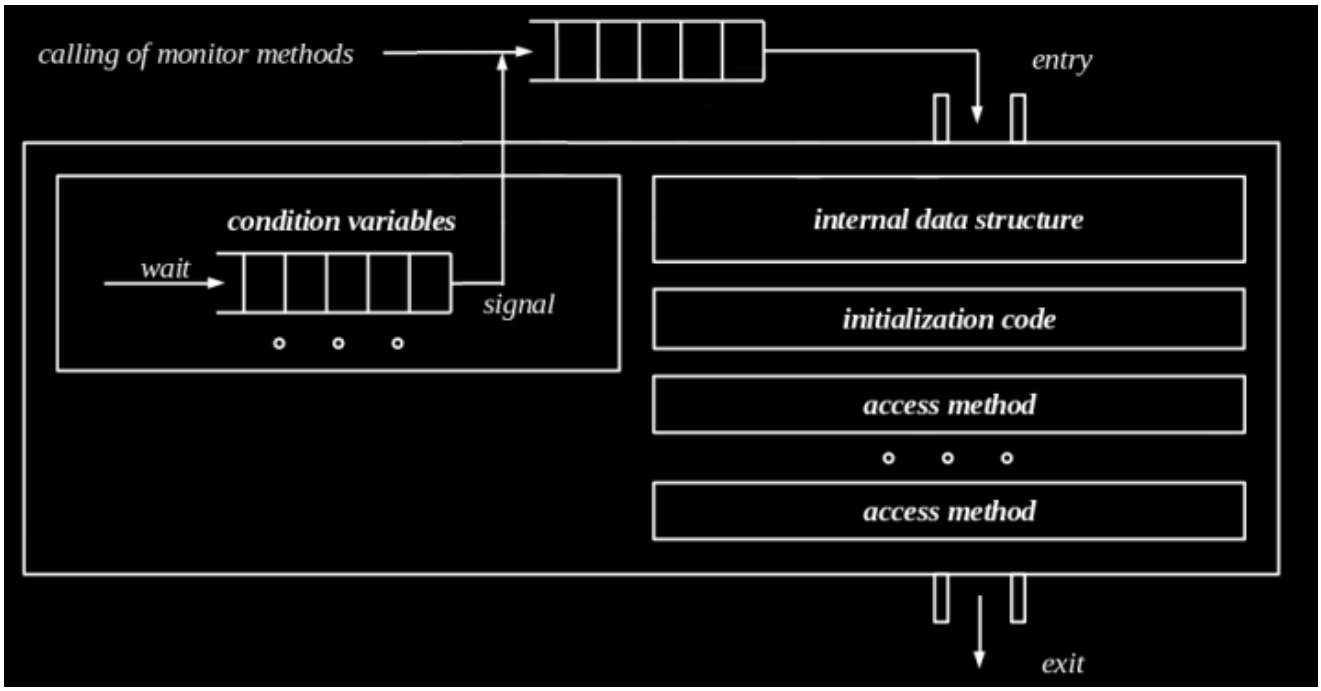
Brinch Hansen monitor - a thread a chamar o signal imediatamente sai do monitor. É facil de implementar, mas restritivo.



Lampson / Redell

O que fazer quando ocorre o `signal`?

Lampson / Redell monitor - a thread a chamar o signal continua a sua execução e a thread acabada de acordar é mantida fora do monitor, competindo por acesso. Facil de implementar, mas pode causar starvation.



Message-passing

Introdução

Processos podem comunicar com troca de mensagens. Utilizam um mecanismo geral de comunicação que nao requer shared memory explicitamente. Inclui comunicação e sincronização. Válido para uniprocessor and multiprocessor systems.

Duas operações necessárias: **send** e **receive**.

É preciso um link de comunicação, que pode ser categorizado em diferentes maneiras:

- Comunicação **direta** ou **indireta**.
- Comunicação **sincrona** ou **assincrona**.
- Tipo de **buffering**.

Comunicação direta e indireta

Comunicação direta *simétrica* - Um processo que quer comunicar deve explicitamente dizer o nome do receiver/sender.

- `send(P, msg)` - mandar mensagem `msg` para processo `P`.
- `receive(P, msg)` - receber mensagem `msg` de processo `P`.

Neste esquema, um link de comunicação tem as seguintes propriedades:

- É estabelecido automaticamente entre um par de communicating processes.
- É associado a exatamente 2 processos.
- Entre um par de communicating processes, existe exatamente um link.

Comunicação direta assimétrica - Apenas o sender deve explicitamente nomear o receiver.

- `send(P, msg)` - mandar mensagem `msg` para processo `P`.
- `receive(id, msg)` - receber mensagem `msg` de qualquer processo.

Comunicação indireta - As mensagens mandadas sao enviadas e recebidas de mailboxes, ou ports.

- `send(M, msg)` - mandar mensagem `msg` para mailbox `M`.
- `receive(M, msg)` - receber mensagem `msg` de mailbox `M`.

Um link de comunicação neste esquema tem as seguintes propriedades:

- É apenas estabelecido se o par de communicating processes tem um mailbox partilhado.
- Pode ser associado entre mais de 2 processos.
- Entre um par de processos, pode existir mais que um link (uma mailbox por cada).

Há problemas em dois ou mais processos a tentar receber mensagens da mesma mailbox.

Sincronização

De um ponto de vista de sincronização, há diferentes opções de design para implementar `send` e `receive`.

Blocking send - O sending process bloqueia ate que a mensagem seja recebida pelo destinatario ou mailbox.

Nonblocking send - O sending process manda a mensagem e continua operação.

Blocking receive - O receiver bloqueia ate que uma mensagem esteja disponivel.

Nonblocking receive - O receiver recebe ou uma mensagem valida ou uma indicação que nao ha mensagens disponiveis.

São possiveis diferentes combinaçoes de send e receive.

Buffering

Há diferentes design options:

- **Zero capacity** - Não há queue. O sender deve bloquear ate que o destinatario receba a mensagem.
- **Bounded capacity** - A queue tem tamanho finito. Se tiver full, o sender deve bloquear ate haver espaço disponivel.
- **Unbounded capacity** - A queue tem potencial length infinito.

Bounded-buffer problem - com mensagens

Os produtores e consumidores partilham mailbox.

O produtor, faz DATA, faz `send` para a mbox.
O consumidor faz `receive`.

Nao é preciso tratar de sincronização/mutex porque as primitivas send e receive tratam disso.

Deadlock

Deadlock

Introdução

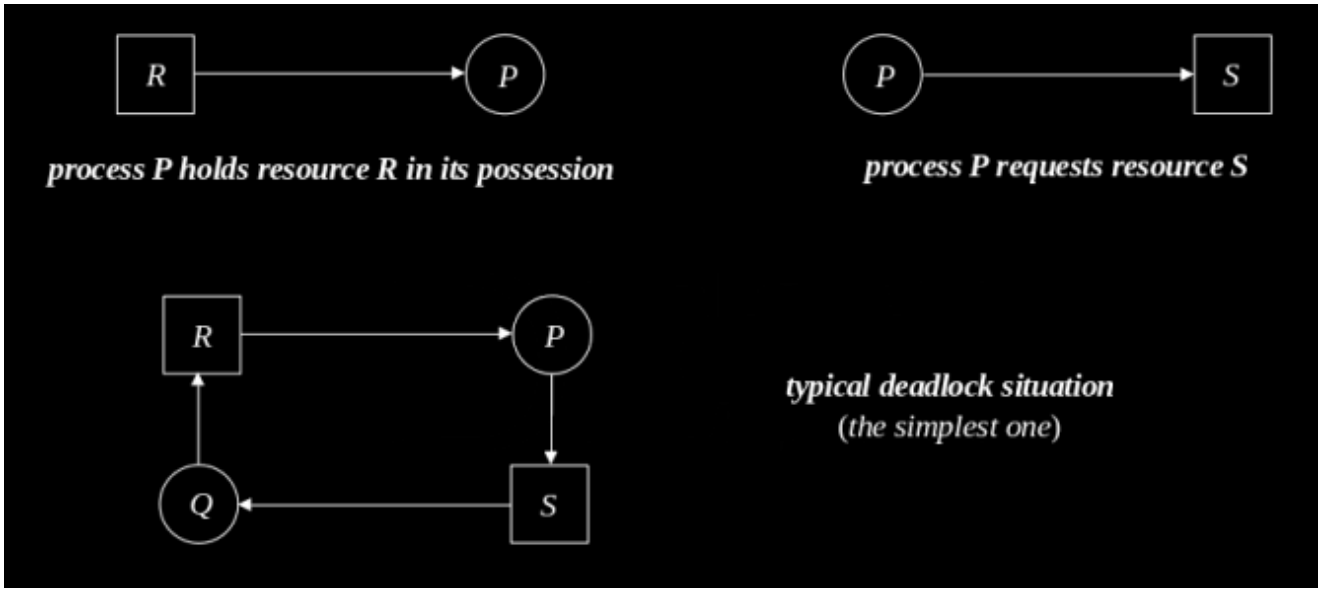
Genericamente, um recurso é algo que um processo precisa para proceder na sua execução.
Recursos podem ser componentes fisicos do computador (processador, memory, I/O...) ou estruturas de dados comuns (PCT, canais de comunicações).

Recursos podem ser:

- **preemptable** - se podem ser withdrawn dos processos que estao a usá-los.
  - e.g processador, regioes de memoria usadas por um process address space.
- **non-preemptable** - se eles apenas podem ser libertados pelos processos que estao a usá-los.
  - e.g um ficheiro, regiao de memoria partilhada que requer acesso exclusivo.

Vamos falar apenas de non-preemptable.

Deadlock ilustrado



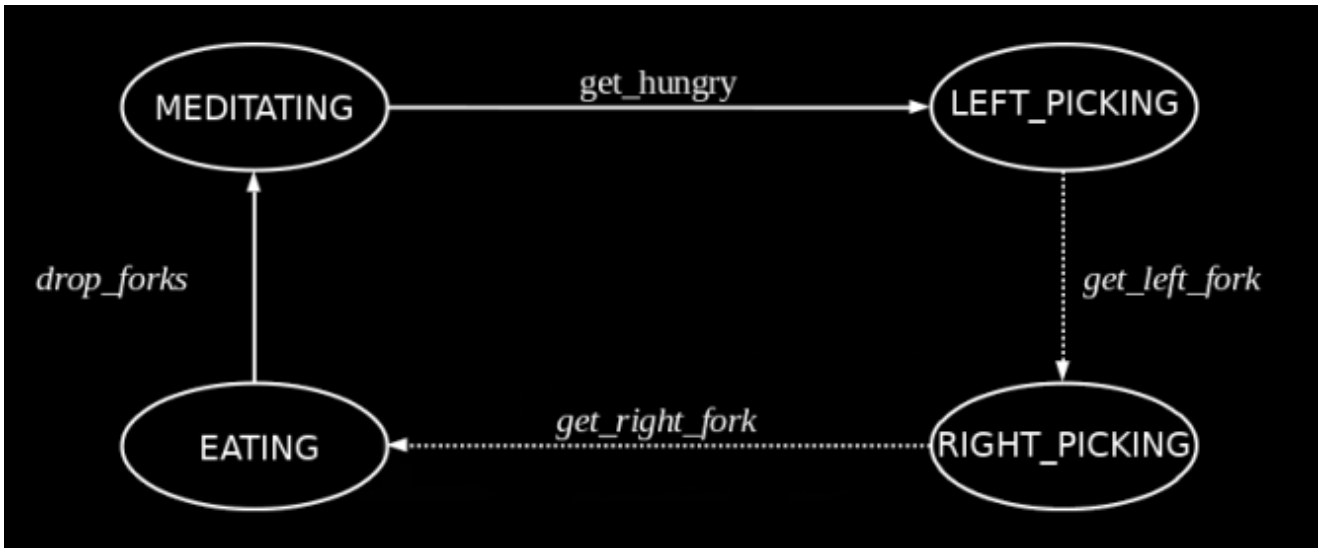
Condições necessárias para deadlock

Pode ser provado que quando deadlock acontece, 4 condições sao sempre observadas.

- **Mutual exclusion** - apenas um processo pode usar um recurso at a time. Se outro processo pede acesso, deve esperar ate que o recurso seja libertado.
- **Hold and wait** - Um processo espera por alguns recursos enquanto tem a posse de alguns.
- **No preemption** - Recursos nao sao preemptable. Apenas o processo com posse do recurso o deve libertar, depois de completar a sua tarefa.
- **Circular wait** - Devem existir processos em que cada um ta a espera de recursos na posse de outros processos. Há loops no grafico.

Philosopher dinner

Solução 1 - Condições de deadlock



Esta solução funciona algumas vezes, mas pode sofrer de deadlock.

Aqui, as 4 condições necessarias previamente tambem sao identificadas:

- **Mutual exclusion** - os garfos nao se podem partilhar ao mesmo tempo.
- **Hold and wait** - Cada filosofo, enquanto fica à espera de obter o garfo da direita, segura o da esquerda.
- **No preemption** - Apenas os filosofos podem libertar os garfos na sua posse.
- **Circular wait** - Se todos os filosofos adquirirem o garfo da esquerda, ha uma chain em que cada filosofo espera pelo garfo de outro filosofo.

Prevenção de deadlock

Definição

Deadlock é **mutual exclusion *E* hold and wait *E* no preemption *E* circular wait**. *Não* deadlock é equivalente a *não* mutual exclusion ***OU*** *não* hold and wait **OU** *não* no preemption ***OU*** *não* circular wait.

Portanto, se podermos mitigar um destes, temos a solução. Chama-se **deadlock prevention.** A aplicação fica responsavel de tratar disto.

Negando as condições necessárias

Negar a condição de mutual exclusion apenas é possivel se recursos podem ser partilhados simultaneamente, else podem ocorrer race conditions.

Negar a condição de preemption é apenas possivel se recursos sao preemptable, que nao costuma acontecer.

Assim, em geral, as unicas outras condições sao usadas para implementar deadlock prevention.

Negar o hold-and-wait pode ser feito se um processo, logo no inicio, pedir todos os recursos que vai necessitar. Pode ocorrer starvation nesta solução. Mecanismos de aging resolvem starvation.

Solução 2

Quando um filosofo quer comer, obtem os dois garfos de uma vez. Se nao tiverem disponivel, tem que esperar. Logo, starvation nao é evitada.

Negando as condições necessárias

Negar o hold and wait tambem pode ser feito. Um processo liberta os recursos ja adquiridos se falha na obtenção de outros recursos. Mais tarde, pode tentar novamente.

Nesta solução, pode acontecer starvation e busy waiting. Aging mechanisms para resolver starvation, e bloquear e acordar processos para busy waiting.

Solução 3

Quando um filosofo quer comer, obtem o garfo da esquerda. Depois, tenta adquirir o garfo da direita. Se não conseguir, liberta o garfo da esquerda e volta ao hungry state. **Busy waiting** e **starvation**.

Negando as condições necessárias

Para negar a circular wait, atribuimos um diferente id a cada recurso e impomos que a aquisição de recursos seja feito de forma ascendente ou descendente. **Starvation** nao evitada.

Solução 4

Cada garfo tem um id, igual ao id do filosofo à sua esquerda. Cada filosofo adquire o garfo com menor id. This way, philosophers 0 to N − 2 acquire first the left fork, while philosopher N − 1 acquires first the right one.

Deadlock avoidance

Definição

Deadlock avoidance é menos restritiva que deadlock prevention

- Nenhuma das condições de deadlock sao negadas a priori.
- O sistema de recursos é monitorizado para que se decida o que se fazer em termos de alocação de recursos.
- Requer conhecimento in advance de maximos requisitos de recursos de um processo. Logo, os processos têm de declarar no inicio os seus requisitos.

Duas approaches possiveis:

- **Process initiation denial** - Nao começar um processo se os seus requisitos possam levar a um deadlock.
- **Resource allocation denial** - Nao dar mais recursos a um processo se essa alocação puder levar a um deadlock.

Process initiation denial

O sistema previne a iniciação de um processo se a sua conclusao nao pode ser garantida.

Um novo processo apenas é começado se a **quantia total dos recursos declarados pelo processo** for *menor ou igual* que a *subtração* da **quantia total de todos os recursos (na posse/declarados por processos)** com a **quantia total de cada recurso**.

É uma approach muito restritiva.

Resource allocation denial

Um novo recurso é alocado a um processo se, e apenas se, ha pelo menos uma sequencia de alocações futuras que nao resultam em deadlock. O sistema está em **safe state**, nesse caso.

formula complicada, skip.

Banker's algorithm

		R1	R2	R3	R4
Total resources		6	5	7	6
Available Resources		3	1	1	2
Resources declared	P1 P2 P3	3 1 1	3 2 3	2 3 5	2 4 0
Resources allocated	P1 P2 P3	1 1 1	2 0 2	2 3 1	1 3 0
Resources requestable	P1 P2 P3	2 0 0	1 2 1	0 0 4	1 1 0

Se o P3 pede 2 R3, leva postpone, porque apenas 1 está disponivel.

Se o P3 pede 1 R2, tambem é postponed, porque se dermos, o sistema fica num **unsafe state**.

Deteção de deadlock

Definição

Nao se usa deadlock prevention ou avoidance, portanto é possivel a ocorrencia de situações de deadlock. O estado do sistema deve ser examinado para determinar se um deadlock ocorreu. Deve existir um procedimento onde se recupera de um deadlock.

Procedimento de recuperação

- **Libertar recursos de um processo** - se for possível
  - O processo é suspendido ate que o recurso possa ser devolvido.
  - Eficiente mas requer a possibilidade de salvar o process state.
- **Rollback** - se os estados de execução de diferentes processos sao periodicamente guardados.
  - Um recurso é libertado de um processo, cujo estado de execução leva roll back para o ultimo "checkpoint".
- **Matar processos** - Radical mas de facil implementação.

Gestão de memoria

Gestão de memoria

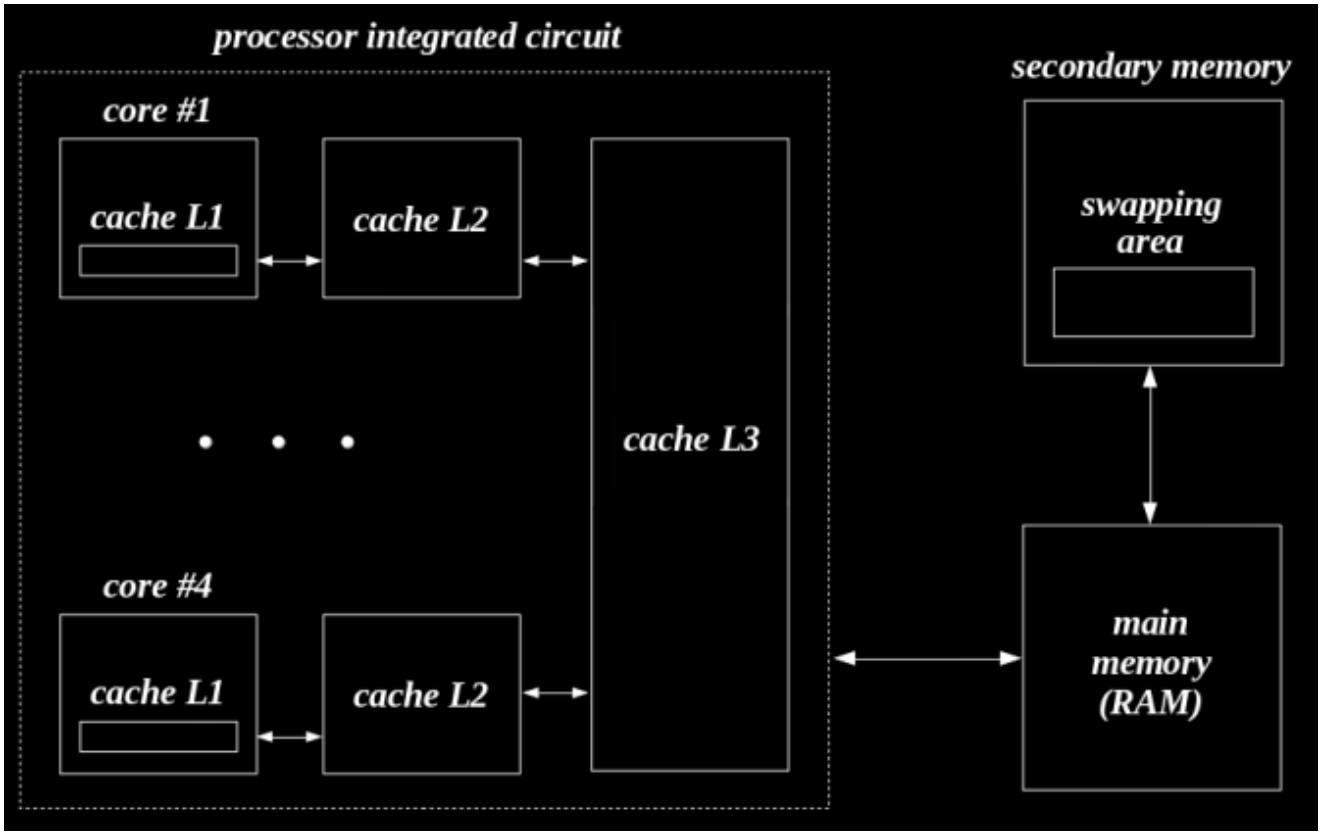
Introdução

Para ser executado, um processo deve ter a sua address space, pelo menos parcialmente, residente na main memory.

Num ambiente multiprogrammed, para maximizar utilização do processador e melhorar response/turnaround time, um sistema deve manter os address spaces de multiplos processos residentes na main memory, mas pode nao haver espaço para todos.

Hierarquia de memoria

- **Cache memory** - Pequena (KB a MB), muito rapida, volatil (mantem dados apenas enquanto o pc ta ligado), e cara.
- **Main memory** - tamanho médio (MB a GB), volatil, preço médio, nao rapida nem lenta.
- **Secondary memory** - tamanho grande (GB a TB), lenta, nao volatil e barata.



A **memoria cache** vai conter uma copia das posições de memoria mais frequentemente referenciadas pelo processador no passado proximo.

- A cache está no circuito integrado do processador (**L1**).
- E num circuito integrado autonomo, colado ao mesmo substrato (**L2** e **L3**).

O programador é abstraido quase completamente da transferência de dados to and from the main memory.

A **memoria secundaria** tem duas funçoes principais:

- **File system** - armazenamento para mais ou menos informação permanente (programas e dados).
- **Swapping area** - Extensão da main memory para que o tamanho nao seja um fator limitante para o numero de processos que possam coexistir. Pode estar num disk partition usado apenas para esse propósito ou ser um ficheiro num file system.

Este tipo de organização é baseado na assumption que, quao mais longe uma instrução tiver do processador, menos vezes sera referenciada.

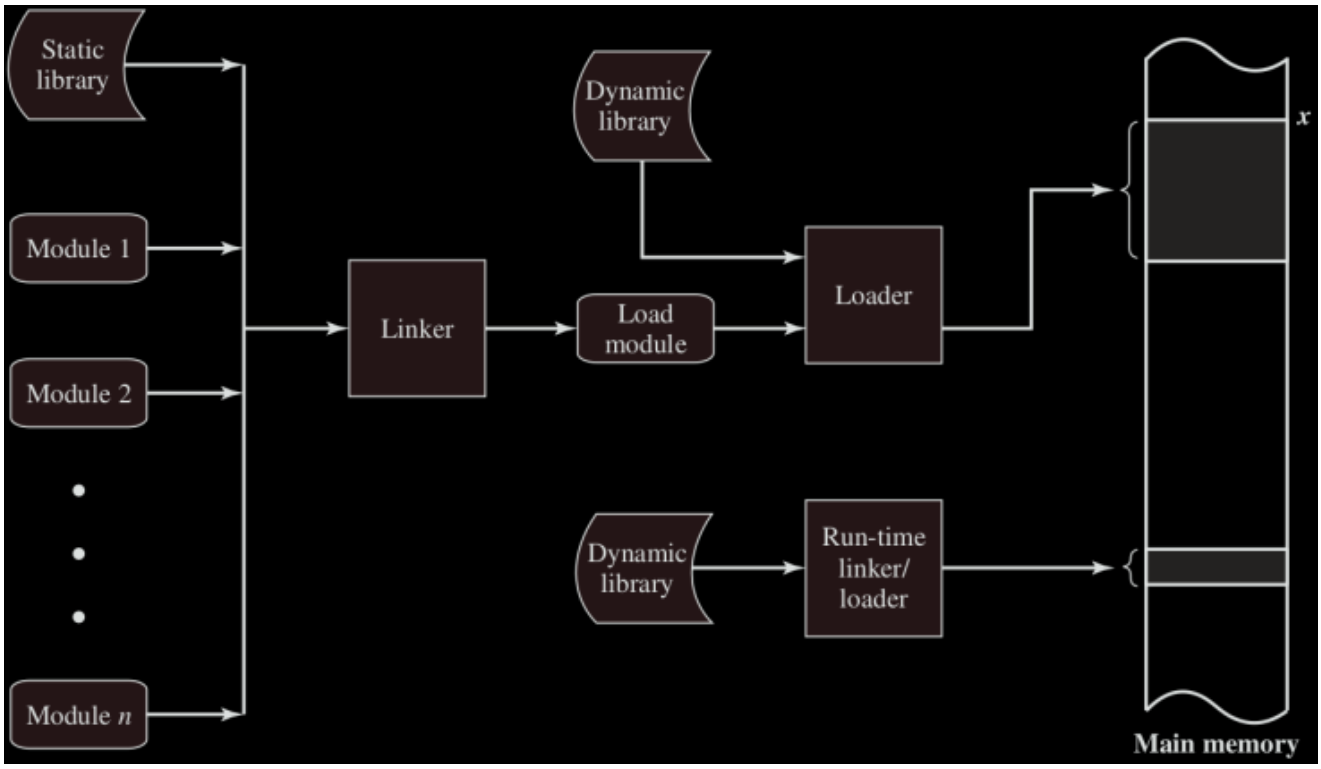
Baseado no principio de **locality of reference**, um programa tende a aceder ao **mesmo set de memory locations repetitivamente** num **curto periodo de tempo**.

Função

A função de gestão de memória num ambiente multiprogrammed foca-se em alocar memoria a processos e a controlar a transferência de dados entre main e secondary memory (**swapping area**), para:

- **Manter um register** das partes da main memory que estão ocupadas e das que tao disponiveis.
- **Reservar porções** da main memory para processos que vao precisar, ou libertá los quando ja nao sao precisos.
- **Dar swap out** a toda ou parte da address space de um processo quando a main memory é muito pequena para conter todos os processos que coexistem.
- **Dar swap in** a toda ou parte da address space de um processo quando main memory fica disponivel.

Funçoes linker e loader



Os **object files**, resultantes do processo de compilaçao, sao files relocatable. As addresses das varias instruções, constantes e variaveis sao calculadas a partir do inicio do modulo, por convenção a address é 0.

A role do processo de **linking** é meter os diferentes object files juntos num file único - o **executable file**, resolving among themselves the various external references.

Bibliotecas estaticas tambem sao incluidas no processo de linking. Bibliotecas **dinamicas** (shared) nao sao.

O **loader** controi a imagem binaria da address space do processo, que vai eventualmente ser executada, combinando o executable file e, se aplicavel, algumas bibliotecas dinamicas, resolving any remaining external references.

Bibliotecas dinamicas tambem podem ser loaded at **run time**.

Quando o *linkage* é dinamico:

- Cada referencia no codigo para uma rotina de uma biblioteca dinamica é substituido por um **stub** - um pequeno set de instruções que determina a localização de uma rotina especifica, se ja é residente na main memory, or promotes its load in memory, otherwise
- Quando um stub é executado, a rotina associada é identificada e localizada na main memory, o stub substitui a referencia para a sua address no codigo do

processo com a address do rotina do sistema e executa a.
- Quando essa zona do código é alcançada outra vez, a rotina do sistema é agora executada diretamente.

Todos os processos que usam a mesma biblioteca dinamica, executam a mesma copia do codigo, assim minimizando a ocupação da main memory.

Object and executable files

Source file

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

int main(void) {
        printf("hello, world!\n");
        exit(EXIT_SUCCESS);
}
```

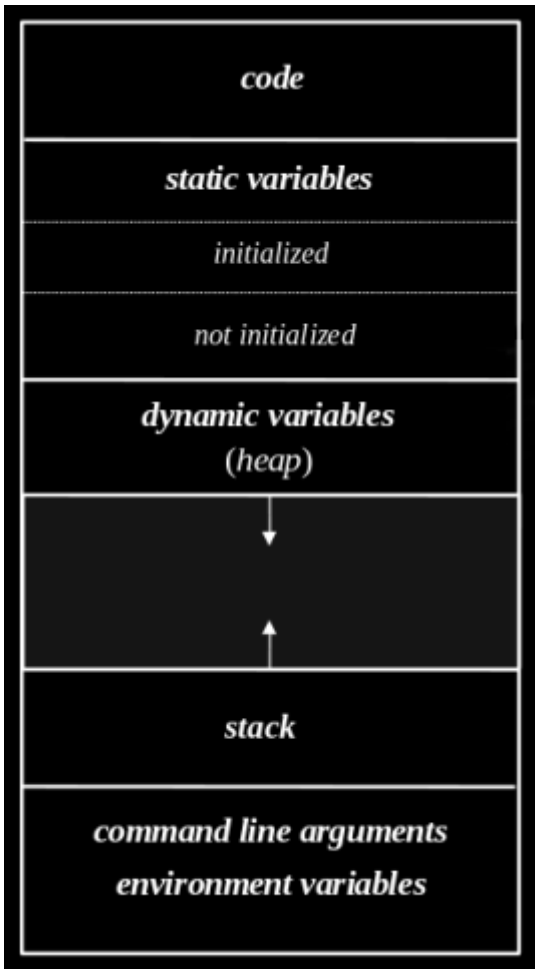
Object file

`gcc -Wall -c hello.c`
`file hello.o`
`hello.o: ELF 64-bit LSB relocatable, x86-64, version 1 (SYSV), not stripped`

Executable file

`gcc -o hello hello.o`
`file hello`
`hello: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically linked, interpreter /1ib64/1d-linux-x86-64.so.2, for GNU/Linux 3.2.0, BuildID[sha1]=48ac0a8ba08d8df6d5e8a27e00b50248a3061876, not stripped`

Address space de um processo



As regioes de **code** e **static variables** tem tamanho fixo, que é determinado pelo loader.
As regioes de **dynamic variables (heap)** e **stack** crescem (um contra o outro) durante a execução do processo.

É pratica comum deixar unallocated memory area na address space do processo entre a regiao de definiçoes dinamicas e o stack. Esta area unallocated pode ser usada entre cada um deles.

Quando esta area é exhausted no stack side, a execução do processo nao pode continuar, resultando em **stack overflow**.

A imagem binaria da address space do processo representa uma address space relocatable, chamada **logical address space**.

A regiao da main memory onde está loaded para execução constitui a **physical address space** do processo.

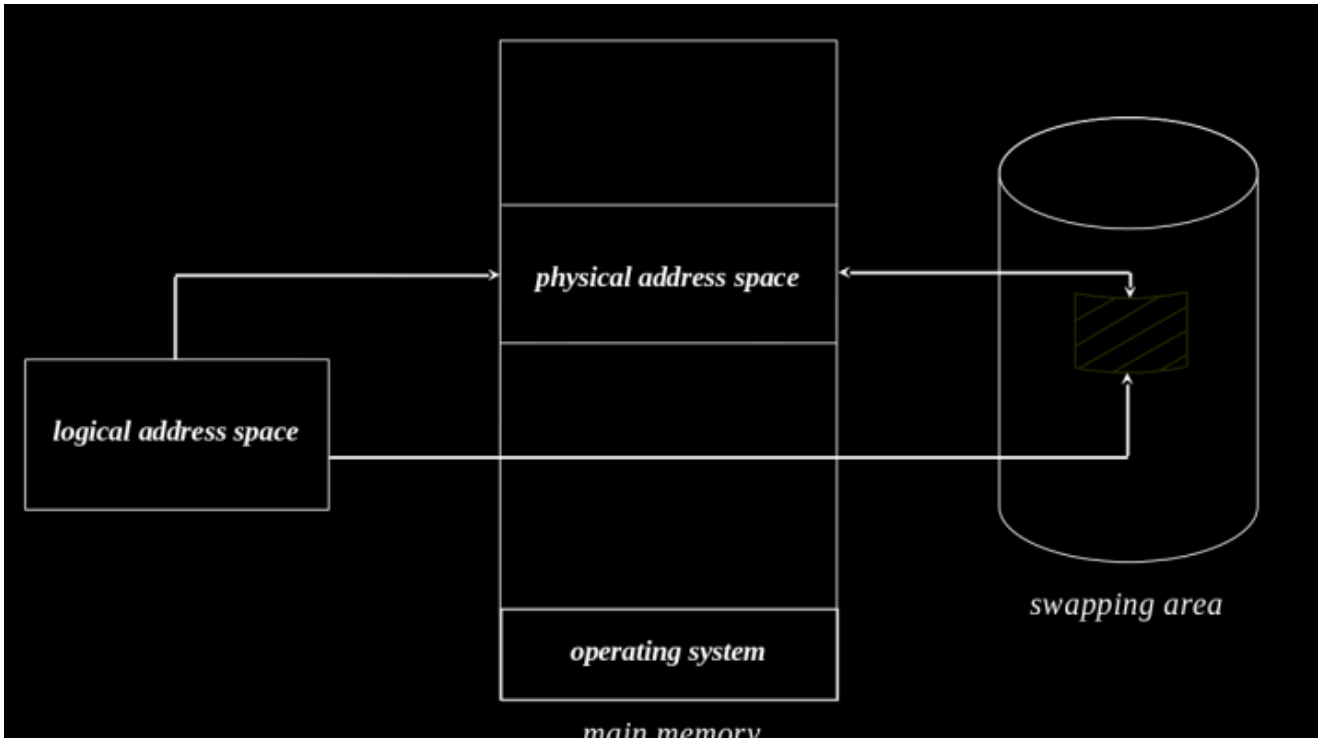
Separação entre as logical e physical address spaces é um conceito central a mecanismos de gestao de memoria.

Ha dois problemas a ser resolvidos:

- **Dynamic mapping** - Habilidade de converter uma logical address para physical address em runtime, para que a physical address space de um processo possa ser colocada em qualquer regiao da main memory e movida se necessario.
- **Dynamic protection** - Habilidade de prevenir acesso em runtime a addresses localizadas fora da address space do processo.

Contiguous memory allocation

Logical and physical address spaces



Em contiguous memory allocation, ha uma correspondencia one-to-one entre a logical address space de um processo e a sua physical address space.

Consequencias:

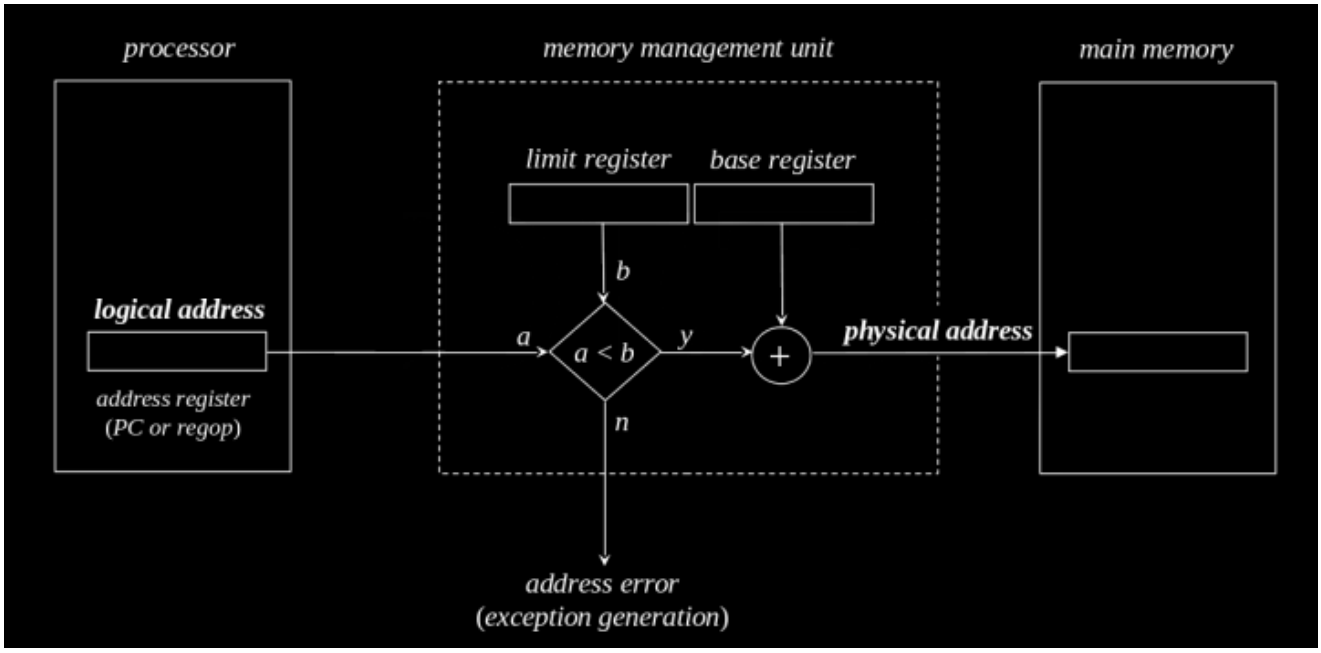
- **Limitação da address space de um processo** - Em nenhum caso é possivel a gestao de memoria suportar mecanismos automaticos que permitem que a address space de um processo seja maior que o tamanho da main memory disponivel.
- **Contiguity of the physical address space** - Apesar de não ser uma condição estritamente necessária, é naturalmente mais simples e eficiente assumir que a address space do processo é contiguous.
- **Swapping area como uma extensao da main memory** - serve para armazenar a address space de processos que nao podem residir na main

memory devido a falta de espaço.

Logical address to physical address translation

Como acontece **dynamic mapping** e **protection**?

- É preciso hardware - *MMU*.



- O **limit register** deve conter o tamanho em bytes do logical address space
- O **base register** deve conter a address do inicio da regiao de main memory onde a physical address space do processo é colocada.
- Em context switching, a operação de *dispatch* dá load aos base e limit registers com os valores presentes nos fields correspondentes da entry da *PCT* associados ao processo que ta a ser scheduled para execução.

Quando ha uma referencia a memoria:

- A logical address é primeiramente comparada ao valor do limit register.
- Se for menor, é uma referencia valida (ocorre dentro do process address space). De seguida, a logical address é adicionada ao valor da base register para produzir o physical address.
- Se for maior ou igual, é uma referencia invalida. De seguida, um null memory access (dummy cycle) é feito e uma exceção gerada devido a address error.

Exemplo:
Logical address -> 0x0010.
Limit register -> 0x0200.

É menor, logo passa
Base register -> 0x1600.

Com a logical address de 0x10, acedemos à address fisica 0x1610. O processo (como está representado contiguamente) vai de 0x1600 a 0x1800. Acho que é assim

Long-term scheduling

Quando um processo é criado, as data structures para gestao sao inicializadas. O seu logical address space é construido e o valor do limit register é calculado e armazenado no campo correspondente da *PCT*.

Se ha espaço na main memory, o seu address space é loaded lá, o campo base register é atualizado com a address inicial da regiao atribuida e o processo é colocado na READY queue.

Se não ha espaço, a sua address space é temporariamente armazenada na swapping area e o processo é colocado na SUSPENDED-READY queue.

Medium-term scheduling

Se memoria é necessaria para outro processo, um processo BLOCKED (ou ate READY) pode levar swap out. Nesse caso, a sua base register no PCT torna-se undefined.

Se memoria se torna disponivel, um processo SUSPENDED-READY (ou SUSPENDED-BLOCKED) pode ser swapped in. Nesse caso, a sua base register no PCT é atualizado com a sua nova physical location. Um processo SUSPENDED-BLOCK apenas é selecionado se nao existem SUSPENDED-READY.

Quando um processo termina, leva swap out, esperando pelo fim das operações.

Partição de memoria

Como fazer

Depois de reservar uma quantia para o sistema operativo, como partir a real memoria para acomodar os diferentes processos?

Fixed partitioning

Main memory pode ser dividida num numero de slices estaticos at system generation time. Nao necessariamente o mesmo tamanho cada slice.

A logical address space de um processo pode ser loaded num slice de igual ou maior size. Assim, a maior slice determina o tamanho do **largest allowable process**.

Features:

- Simples para implementar.
- Eficiente - pouco overhead.
- Numero fixo de processos allowable.
- Uso ineficiente de memoria devido a fragmentaçao interna. A parte de uma slice nao usada por um processo é gasta.

Se uma slice se torna disponivel, qual dos processos SUSPENDED-READY devem ser colocados lá?

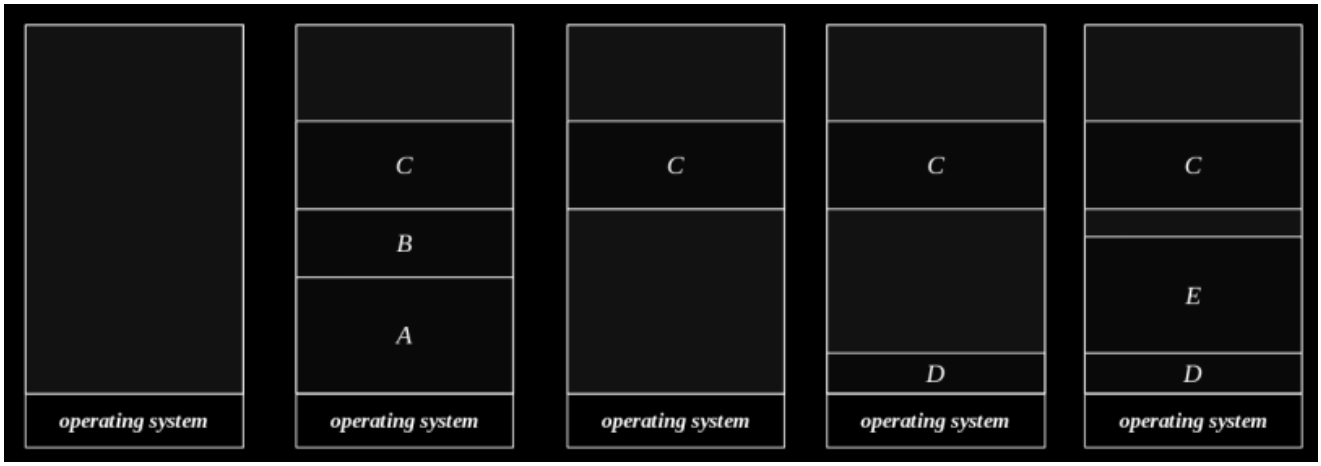
Duas diferentes politicas de scheduling sao consideradas aqui:

- **Valorizando justiça** - O primeiro processo na queue de SUSPENDED-READY cuja address caiba na slice é escolhido.

- **Valorizando a ocupação de main memory** - O primeiro processo na queue de SUSPENDED-READY com o maior address space que caiba na slice é escolhido. Para evitar starvation, utiliza-se um aging mechanism.

Dynamic partitioning

Em dynamic partitioning, no começo, todas as partes disponiveis da memoria constituem um unico bloco e depois reservam regioes de tamanho suficiente para dar load à address space de processos que apareçam. Libertar estas regioes quando ja nao sao precisas.



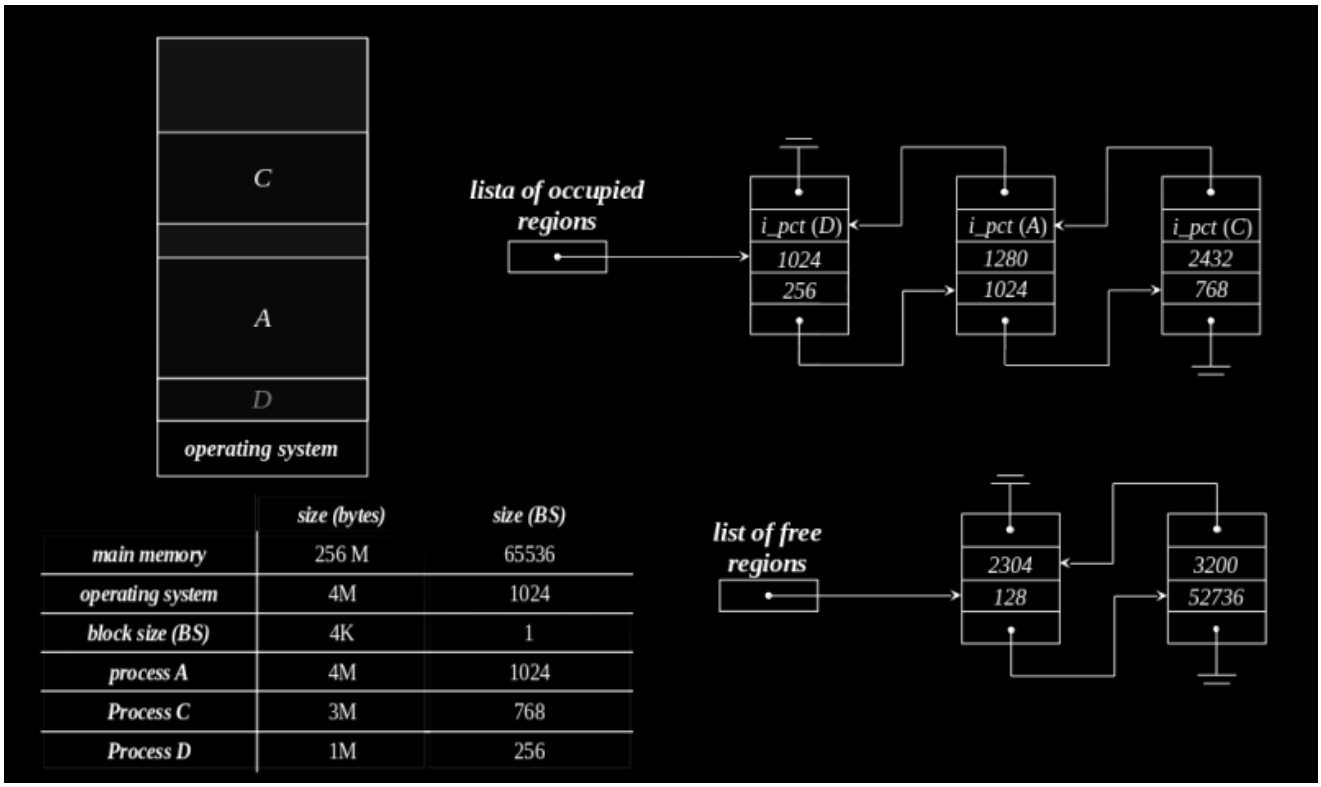
Como a memoria é dinamicamente reservada e libertada, o sistema operativo tem que manter um historico atualizado de regioes ocupadas e libertadas.

Uma forma de fazer isto é construindo duas bilinked lists (ha atributo previous, alem do next). Uma para **regioes ocupadas** e outra para **regioes livres**.

Memoria nao é alocada em *byte boundaries*, porque:

- Podem aparecer regioes disponiveis muito pequenas, inuteis.
- Sera incluida na lista de regioes livres.
- Faz searches mais complexas.

Assim, a main memory é tipicamente dividida em blocos de tamanho fixo e alocação é feita em unidades destes blocos - *chunk size?*.



	size (bytes)	size (BS)
main memory	256 M	65536
operating system	4M	1024
block size (BS)	4K	1
process A	4M	1024
Process C	3M	768
Process D	1M	256

Na lista, temos cada processo, o seu tamanho da address space e o começo.

Valorizar justiça é a politica geralmente usada, ou seja, escolhe-se o primeiro processo na queue de SUSPENDED-READY.

Dynamic partitioning pode produzir external fragmentation.

- Espaço livre é dividido num grande numero de possivelmente regioes livres pequenas.
- Situações podem ser alcançandas onde, apesar de haver memoria livre suficiente, nao é continua, e o armazenamento da address space de um processo nao é possivel.

A solução é **garbage collection** - compactar o espaço livre, agrupando todas as regioes livres numa unica. - Esta operação necessita de pausar todo o processamento e, se a memoria é larga o suficiente, pode ter um tempo de execução muito grande.

No caso da existencia de muitas regioes livres disponiveis, qual usar para alocar a address space?

Politicas:

- **first fit** - a lista de regioes livres é pesquisada desde o inicio ate ao fim. A primeira regiao com tamanho suficiente é usada.

- **next fit** - é como o first fit, mas começa a pesquisar na slice onde parou na ultima iteraçao.
- **best fit** - a lista de regioes livres é fully searched. Escolhe-se a regiao mais pequena com tamanho suficiente.
- **worst fit** - best fit, mas escolhe-se a maior regiao.

Vantagens:

- **Geral** - A scope das aplicações nao depende do tipo de processos que vao ser executados.
- **Implementação de baixa complexidade** - nao é necessario hardware especial e as estruturas de dados sao reduzidas a duas listas biligadas.

Desvantagens:

- **Fragmentação externa** - a fraçao de memoria que acaba por ser gasta pode chegar, em alguns casos a **1/3** do total.
- **Ineficiente** - nao é possivel construir algoritmos que sao simultaneamente muito eficientes em alocar e libertar espaço.

Virtual memory system

Mapping of the logical address space

Num sistema de memoria virtual, a logical address space do processo e o sua physical address space sao totalmente disassociados.

A logical address space está dividida em diferentes blocos, para que consigam espalhar-se pela physical address space.

Um processo pode estar apenas parcialmente residente na main memory, portanto alguns dos seus blocos podem estar apenas na swapping area.

Algumas features

Non-contiguity of the physical address space - as address spaces de processos, divididas em blocos de tamanho fixo ou variado, estao espalhadas pela memoria, tentando garantir uma ocupaçao mais eficiente do espaço disponivel.

Nao ha limitação da address space de um processo - metodologias que permitem execução de processos cujos address spaces sao maiores do que o

tamanho da main memory disponivel podem ser estabelecidos.

Swapping area como uma extensao da main memory - a sua função é manter uma imagem atualizada dos address spaces dos processos que atualmente coexistem, principalmente a parte de variaveis (static e dynamic definition e stack).

Logical address to physical address translation

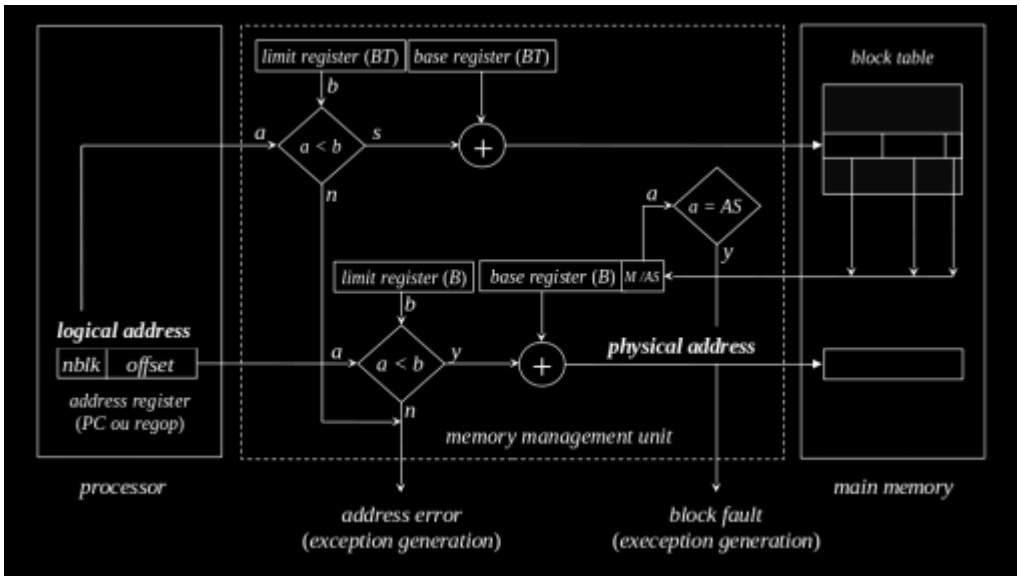
Como é feito dynamic mapping e protection?

Uma logical address é feita de 2 partes:

- **nblk** - identifica um logical block especifico
- **offset** - identifica uma posiçao dentro de um bloco, como um offset desde o seu inicio.

Uma **block table**, armazenada em memoria, mapeia cada bloco logico à sua counterpart fisica.

A **MMU** trata desta estrutura.



A MMU deve conter dois pares registers de base e limit.

- Um par (**BT**) está relacionado com o inicio e tamanho da block table, é uma estrutura de dados que descreve os varios blocos que a logical address space tem.
- O outro par (**B**) descreve um bloco em especifico, o que está a ser acedido.

Em context switching, a operação **dispatch** dá load aos relacionados à block table, com os valores armazenados na entrada PCT do processo scheduled para

execução.

O **BT base register** representa a start address da process block table.
O **BT limit register** representa o numero de entradas da table (numero de blocos).

Um acesso à memoria divide-se em 3 passos.

Passo 1:

- O **nblk** da logical address é comparado com o **BT limit register**.
- Se for valido (menor ou igual), o **BT base register** mais o **nblk** apontam para a block table entry, que é loaded para a MMU.
- Se nao for valido, dá null memory access (dummy cycle) e é gerada uma exceção de address error.

Passo 2:

- a flag **M/AS** é avaliada
- Se for M (o bloco estar referenciado na memoria), a operaçao pode proceder.
- Se nao for (ta swapped out), dá null memory access (dummy cycle) e é gerada uma exceção de block fault.

Passo 3:

- O campo **offset** da logical address é comparado com o **B limit register**.
- Se for valido (menor ou igual), o **B base register** mais o **offset** apontam para a physical address.
- Se nao for, dá null memory access e é gerada uma exceção de address error.

Analise

O aumento em versatilidade, introduzido pelo sistema de memoria virtual tem o custo de transformar cada acesso à memoria em 2 acessos.

- No primeiro, a process' block table é acedida, para obter a address do inicio do bloco na memoria.
- Só no segundo passo é que a posiçao especifica de memoria é acedida.

Conceptualmente, o sistema de memoria virtual resulta na partição do logical address space do processo em blocos que sao dinamicamente tratados como autonomous address sub-spaces numa alocação contigua de memoria.

- De partições fixas, se os blocos sao do mesmo tamanho.
- De partiçoes que variam, se têm tamanho diferente.

O que é novo, é a possibilidade de aceder a um bloco que atualmente nao reside na main memory, com a necessidade de cancelar o acesso e repeti lo mais tarde, quando o bloco fica loaded.

Funçao da TLB

A necessidade deste duplo acesso à memoria pode ser minimizado, tomando vantagem do principio de **locality of reference**.

Como um programa tende a aceder ao mesmo set de memory locations repetitivamente num curto periodo de tempo, os acessos sao concentrados num set de blocos bem definidos.

Então a MMU costuma manter o conteudo das entradas da block table armazenados numa memoria associativa interna, chamada **translation lookaside buffer (TLB)**.

Assim, o primeiro acesso pode ser um:

- **hit** - quando a entrada está armazenada na TLB, o acesso é feito na MMU.
- **miss** - quando a entrada nao está armazenada na TLB, o acesso é feito na main memory.

O average access time é um acesso à TLB mais um acesso à main memory.

Long-term scheduling

Quando um processo é criado, as estruturas de dados que o gere sao inicializadas.

- O seu logical space address é construido, pelo menos a sua parte de variaveis é posto na swapping area e a sua block table é organizada.
- Alguns blocos podem ser partilhados com outros processos.

Se ha espaço na main memory, pelo menos na block table, o primeiro bloco de codigo e bloco da sua stack sao loaded lá, as entradas correspondentes na block table sao atualizadas e o processo é colocado na READY queue, else é colocado na suspended-ready queue.

Short-term scheduling

Durante a sua execução, quando ha **block fault**, um processo é colocado no BLOCKED state, enquanto o faulty block é swapped in. Quando o bloco está em memoria, o processo é metido no READY state.

Medium-term scheduling

Enquanto READY ou BLOCKED, todos os blocos de um processo devem ser swapped out, se memory space é necessario.

Enquanto SUSPENDED-READY (ou SUSPENDED-BLOCKED), se memoria fica disponivel, a block table e uma seleçao de blocos de um processo podem ser swapped in, e o processo fica em READY ou BLOCKED. As entradas correspondentes da block table sao atualizadas. Um processo SUSPENDED-BLOCK é apenas selecionado se nao ha suspended-readys.

Block fault exception

Uma propriedade relevante do sistema de memoria virtual é a capacidade do computador executar processos cujos address spaces nao estao, inteiramente, e simultanemante, residindo na main memory.

O sistema operativo tem que fornecer meios para resolver o problema de referenciar uma address localizada num bloco que nao está presente na memoria.

A MMU gera uma exceção nessas circunstancias e a sua service routine deve começar açoes para fazer swap in ao bloco e, depois da conclusao, repetir a execução da instrução que produziu a fault.

Todas estas operações sao executadas de uma forma totalmente transparente ao utilizador.

Procedimento do serviço

O contexto do processo é armazenado na entrada correspondente da PCT, o seu estado trocado para BLOCKED e o program counter é atualizado com a address que produziu o block fault.

Se memory space fica disponivel, uma regiao é selecionada para dar swap in do missing block, senao um bloco ocupado é selecionado para ser substituido. Se este bloco foi modificado, é transferido para a swapping area. A sua entrada na block table é atualizada para indicar que nao está na memoria.

O swapping in do missing block começa.

A função dispatch do scheduler é chamada para meter em execução um dos processos READY.

Quando a transferencia está concluida, a entrada da block table é atualizada e o processo é posto em READY state.

Analise

Se, durante a sua execução, um processo foi continuadamente gerando block fault exceptions, a taxa de processamento seria muito lenta e, em geral, o throughput do computador seria tambem muito baixo.

Na pratica, isto nao é o caso, por causa do principio da **locality of reference**.

Paging

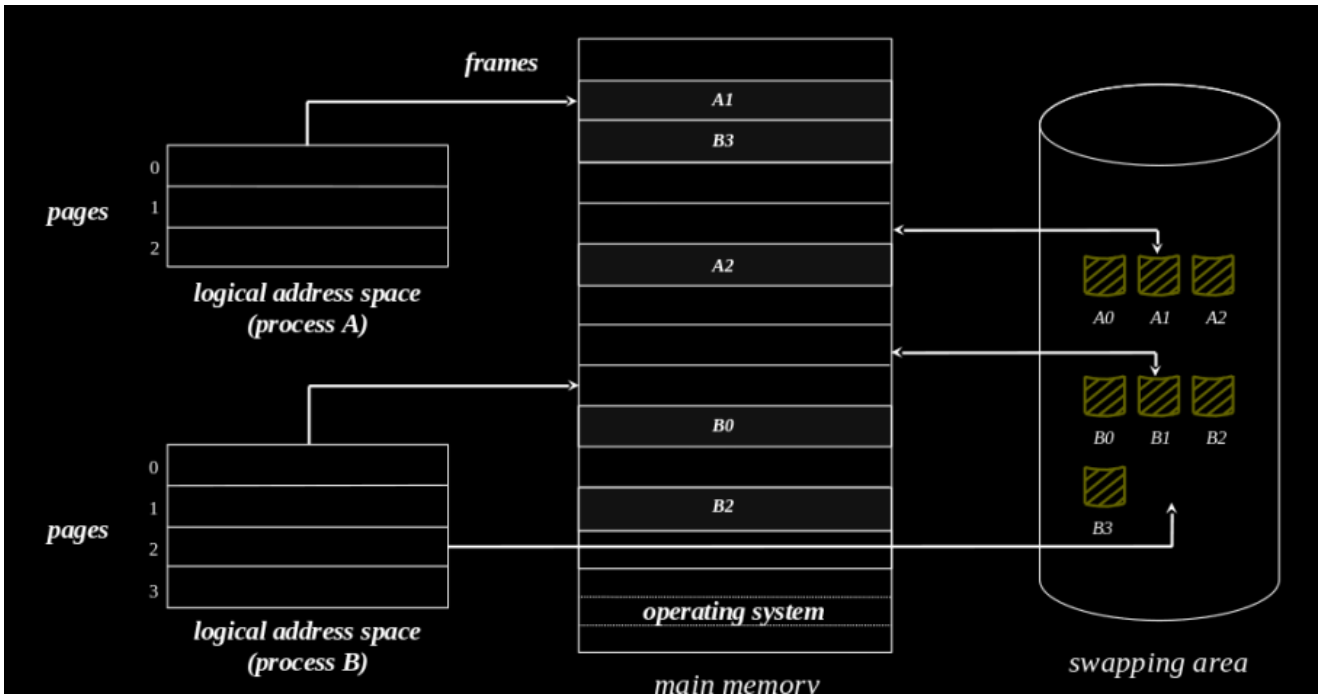
Introdução

A memória está dividida em chunks de tamanho fixo igual, chamados **frames**. É usado um power of 2 para o tamanho, tipicamente 4 ou 8 KB.

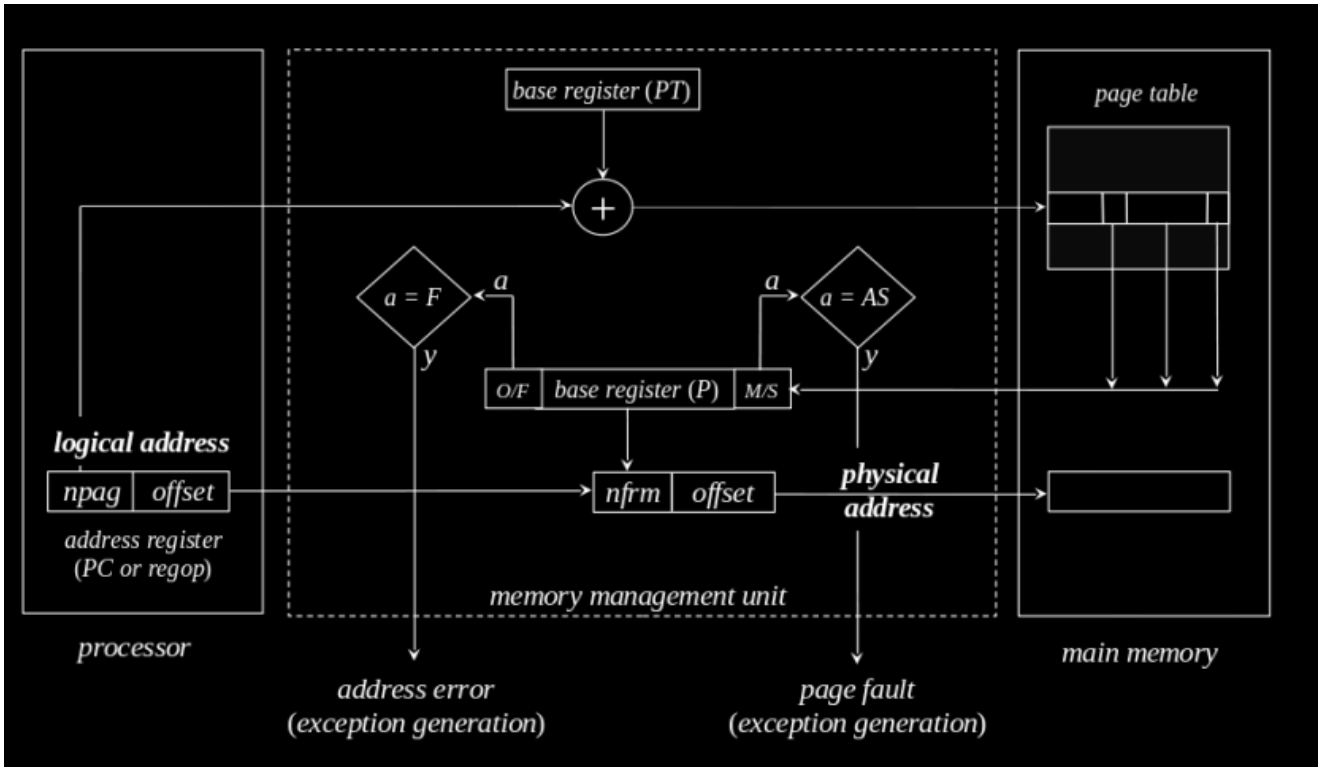
A logical address space de um processo é dividido em blocos de tamanho fixo, do mesmo tamanho, chamados **pages**.

Enquanto se divide a address space em pages, o linker costuma começar uma nova pagina quando um novo segmento começa.

Numa logical address, os bits mais significativos representam o page number e os menos representam o offset dentro da pagina.



Memory management unit (MMU)



Logical address to physical address translation

Estruturando o logical address space do processo para mapear tudo, ou pelo menos uma fraçao, da address space fornecida pelo processador (sempre maior ou igual ao tamanho da main memory existente), torna-se possivel eliminar a necessidade do **limit register** associado com o tamanho da page table. Como consequencia, o *gap* entre a heap memory e a stack pode ser maximizado.

Não há limit register associado com a page.

Entradas da page table

A Page table contem uma entrada por page.

O/F	M/S	ref	mod	perm	frame number	block number in swap area

- **O/F** - flag que indica se a page ja foi associada ao processo
- **M/S** - flag que indica se a page está em memoria.
- **ref** - flag que indica se a page ja foi referenciada.
- **mod** - flag que indica se a page ja foi modificada.
- **perm** - permissoes.
- **block number in swap area** - bloco onde esta a page, na swapping area.

Analise

Vantangens:

- **Geral** - A scope da aplicação nao depende do tipo de processos que vao ser executados (numero e tamanho dos seus address spaces).
- **Bom uso de main memory** - nao leva a fragmentaçao externa e fragmentaçao interna é praticamente negligible.
- **Nao tem requisitos de hardware especial** - As unidades de gestao de memoria nos processadores gerais atuais implementam-o.

Desvantagens:

- **Acesso a memoria mais longo** - Duplo acesso a memoria, por causa de previo acesso à page table. Existencia de TLB minimiza impacto.
- **Operability muito demanding** - requer a existencia de um set de operaçoes de suporte, que sao complexos e têm de ser cuidadosamente feitos para nao comprometer eficiencia.

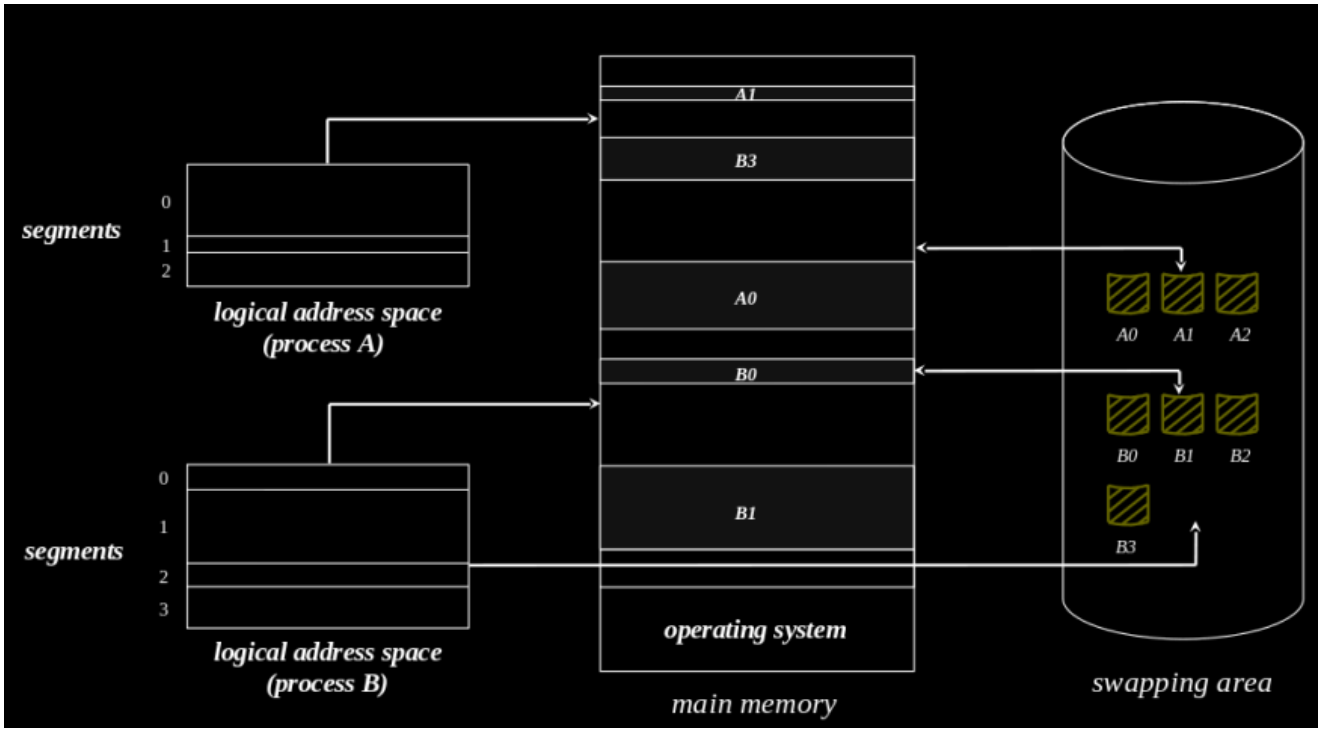
Segmentation

Introdução

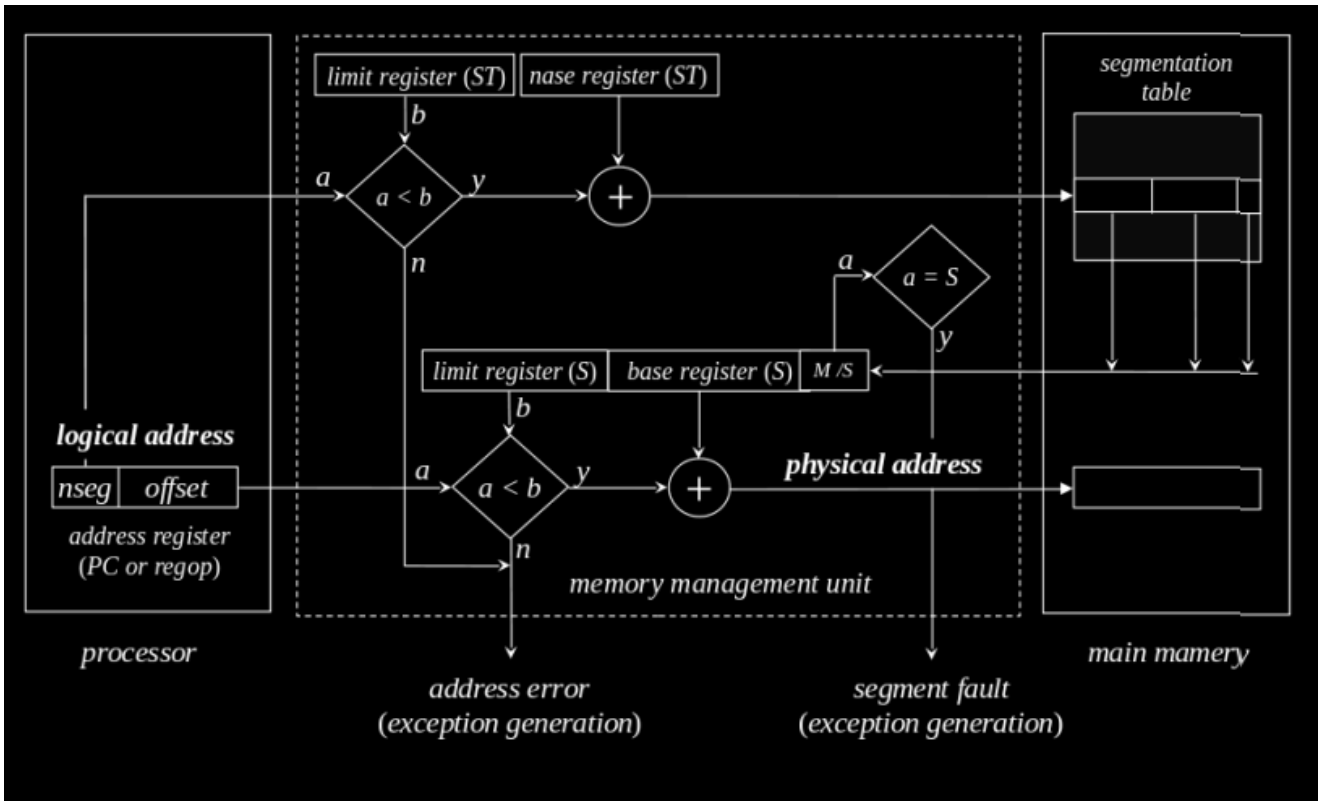
Tipicamente, a logical address space de um processo é composto por diferentes tipos de segmentos:

- **code** - um segmento por cada modulo de codigo.
- **static variables** - um segmento por modulo contendo static variables
- **heap memory** - um segmento
- **shared memory** - um segmento por regiao partilhada
- **stack** - um segmento
- Diferentes segmentos podem ter diferentes tamanhos.

Numa arquitetura de segmentação, os segmentos de um processo sao manipulados separadamente. Partição dinamica pode ser usado para alocar cada segmento. Como consequencia, um processo pode nao ser continuo na memoria. Alguns segmentos podem ate nem estar na main memory.



Memory management unit



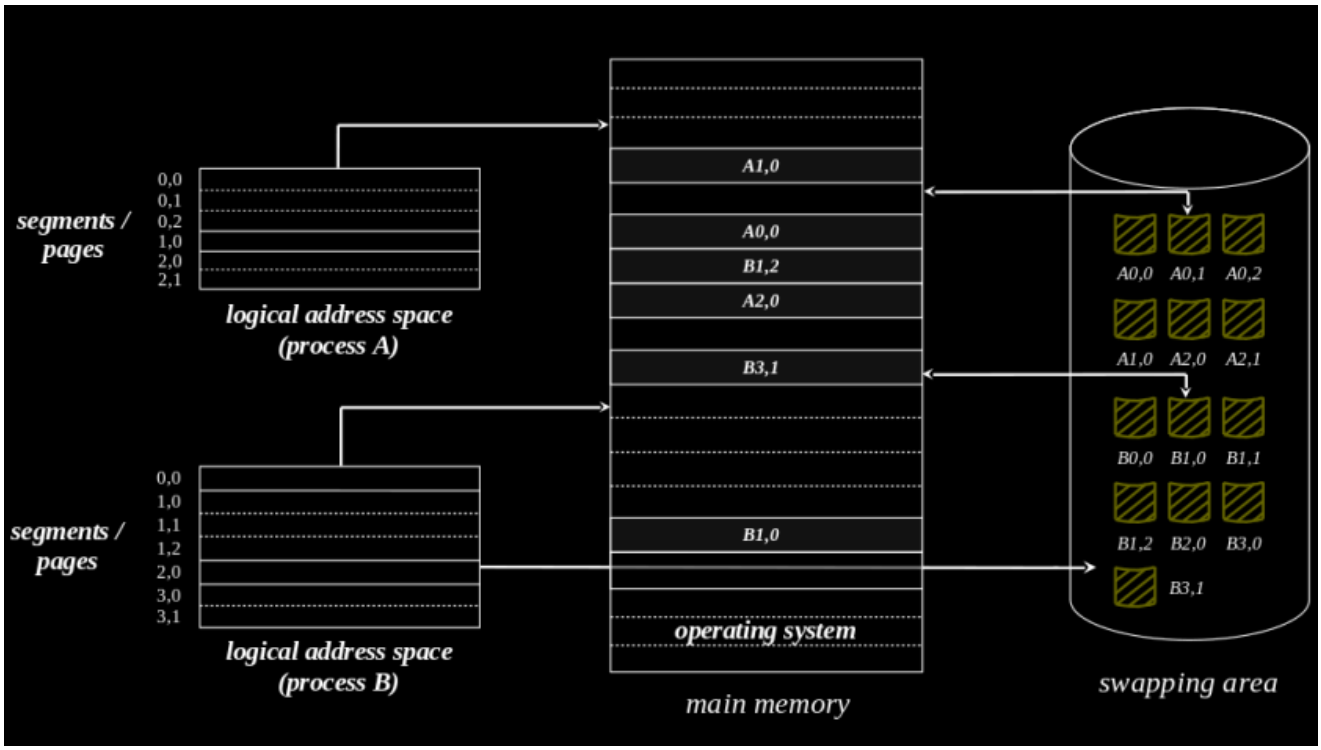
Identico ao com block table

Combinando segmentation e paging

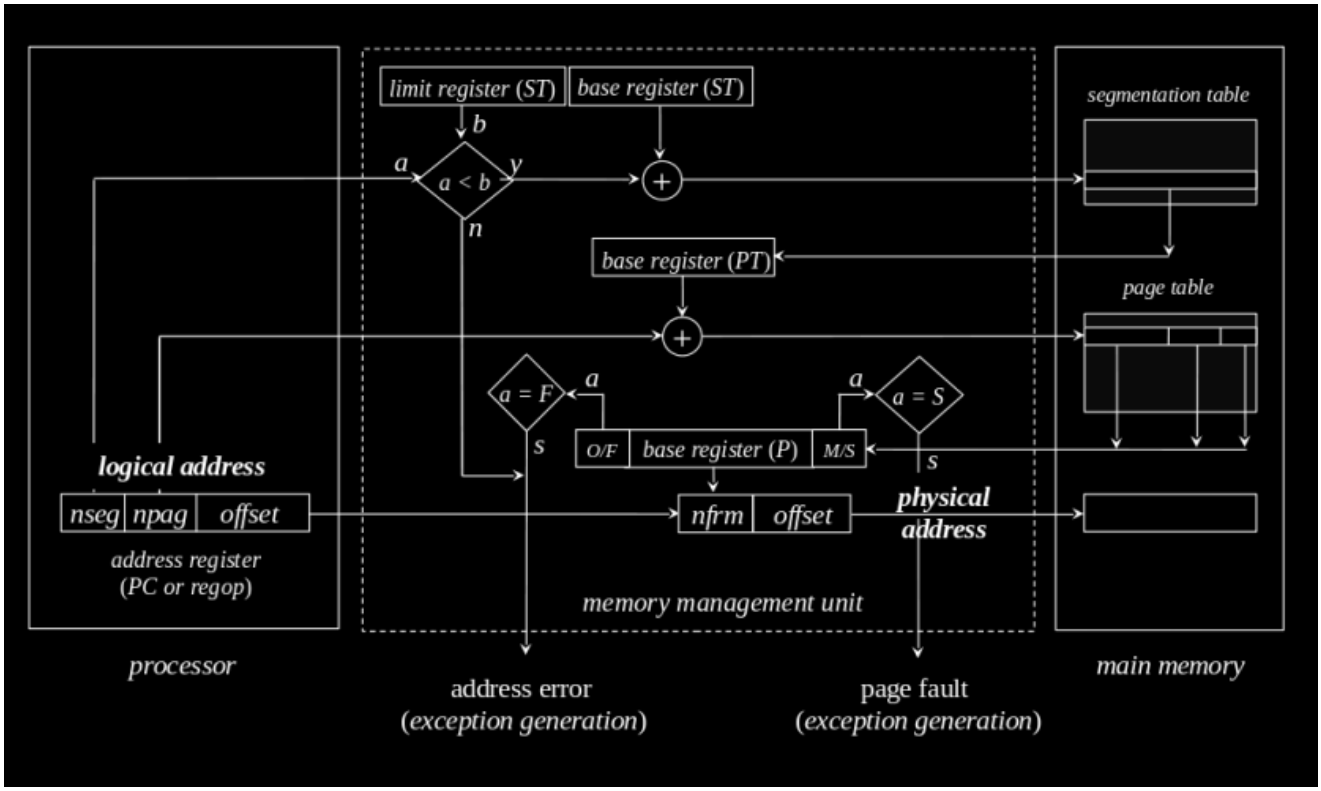
Segmentation sozinho pode ter drawbacks: pode resultar em fragmentaçao externa; Um segmento a crescer pode impor uma mudança na sua localizaçao.

Fundir segmentation e paging pode resolver estes problemas. Primeiro, o logical address space de um processo é partido em segmentos. Depois cada segmento é dividido em pages.

Isto introduz uma complexidade acrescida.



Memory management unit



Logical address to physical address translation

A MMU deve conter 3 base registers e 1 limit register

- 1 base register para a segmentation table
- 1 limit register para a segmentation table
- 1 base regsiter para a page table

- 1 base register para a memory frame

Acesso à memoria divide-se em 3 passos:

- Acesso à segmentation table, acesso à page table e acesso à physical address.

Entrada da segmentation table:

perm	memory address of the page table

Entrada da page table:

O/F	M/S	ref	mod	frame number	block number in swap area

O field **perm** agora é associado ao segmento.

Substituição de page

Introdução

Numa arquitetura de paging (ou combinaçao de segmentation e paging), memoria é partida em frames, tendo cada o mesmo tamanho de uma page. Uma frame pode estar ou free ou ocupada (contendo uma page).

Uma page em memoria pode estar *locked* - se nao pode ser removida de memoria (kernel, buffer cache, ...), ou *unlocked* - pode ser removida de memoria.

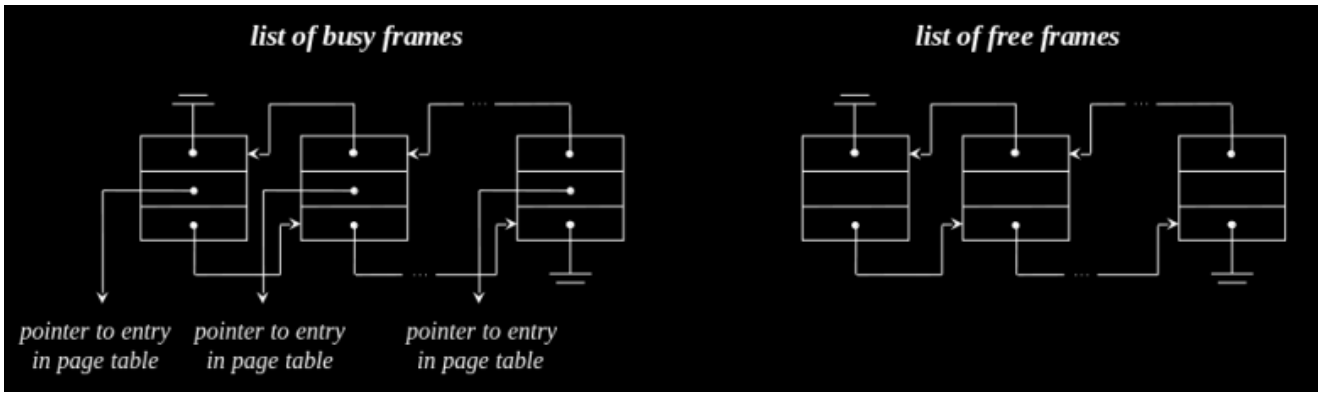
Se nao ha nenhuma frame livre, uma ocupada pode ser libertada. Este é o proposito de **page replacement**. **Apenas se aplica a unlocked pages.**

Lista de frames livres e ocupadas.

Frames livres estao ocupadas numa lista - lista de frames livres.

Frames ocupadas, associadas a pages unlocked, sao tambem organizadas numa lista - lista de frames ocupadas.

A forma de organização da lista de frames ocupadas depende da **politica de page replacement**.



Action on page fault

Em page fault, se a lista de frames livres está vazia, uma frame ocupada deve ser selecionada para ser substituida.

Alternativamente, o sistema pode promover page replacement para manter a lista de frames livres sempre com alguns elementos. Isto permite dar load à faulty page e libertar uma frame ocupada ao mesmo tempo.

A questão é: qual frame deve ser selecionado para substituiçao?

Uma politica optimal seleciona a page cujo tempo de referencia é o maior, mas isto é impossivel de implementar, porque nao sabemos qual é.

Politicas de substituiçao de pages

Least Recently Used

A politica LRU seleciona a frame that has not been referenced the longest. Baseado no principio de **locality of reference**.

Cada frame deve ser labelled com o tempo da ultima referencia. Hardware especifico adicional pode ser necessario.

Em page replacement, a lista de frames ocupadas deve ser iterada para encontrar a que tem o access time mais velho.

Alto custo de implementação e nada eficiente.

Not Recently Used

Os Bits Ref e Mod, campos da entrada da page table, e tipicamente processados por uma MMU convencional, sao usados para definir classes de frames.

class	Ref	Mod
0	0	0
1	0	1
2	1	0
3	1	1

Em page replacement, o algoritmo seleciona aleatoriamente uma frame da lowest non-empty class.

Periodicamente, o sistema itera a lista de frames ocupadas e poe Ref a 0.

Fifo

A politica FIFO seleciona a page baseado no tempo que fica em memoria. Baseado na assumption que, quanto mais tempo uma page residir em memoria, menos likely é ser referenciada no futuro.

A lista de frames ocupadas é considerada ser organizada numa FIFO que reflete a loading order.

Em page replacement, a frame com oldest page é selecionada.

A assumption é extremamente falivel.

Second chance

Esta politica é um upgrade do algoritmo FIFO, dando uma segunda chance à page antes de ser substituida.

Em page replacement:

- A frame com page mais velha é selecionada como **candidata**.
- Se o seu bit **Ref** está a zero, a seleção é feita.
- Senão, o Ref da frame candidata leva reset, a frame é inserida outra vez na fifo, e o processo procede para a proxima frame.
- O processo acaba quando uma frame tem o Ref == 0.

A frame é sempre encontrada.

Clock

A politica clock é um upgrade do algoritmo de second chance, evitando a remoçao e reinserção de elementos na FIFO.

A lista é transformada numa circular e um pointer assinala o elemento mais velho. A ação de remoção seguido da de reinserção corresponde ao avanço do pointer.

Em page replacement:

- Enquanto o Ref está diferente de 0, esse bit leva reset e o pointer avança para a proxima frame.
- A primeira frame com o Ref igual a 0 é escolhido pa replacement.
- Depois de replacement, o pointer é aponta para o proximo elemento.

Working set

Assume que inicialmente, apenas 2 pages de um processo estao em memoria. A page contendo a primeira instrução e a page contendo o inicio do stack.

Page faults serao frequentes, assim que comece a execução. Depois, o processo vai entrar numa fase em que page faults vao ser quase inexistentes, por causa do principio de locality of reference.

Este set de pages chama-se o working set do processo.

À medida que passa o tempo, o working set do processo vai variar, nao apenas em numero, mas tambem com pages especificas que o definem.

Thrashing

Considera que o numero maximo de frames atribuido a um processo é fixo.

Se este numero é sempre maior ou igual ao numero de pages dos diferentes working sets do processo, a vida do processo sera uma sucessao de periodos com page faults frequentes com periodos quase sem eles. Se for menor, o processo vai continuadamente gerar page faults. Nesses casos, chama-se **thrashing**.

Demand paging vs prepaging

Quando um processo troca para o ready state, que pages devem ser colocadas na main memory?

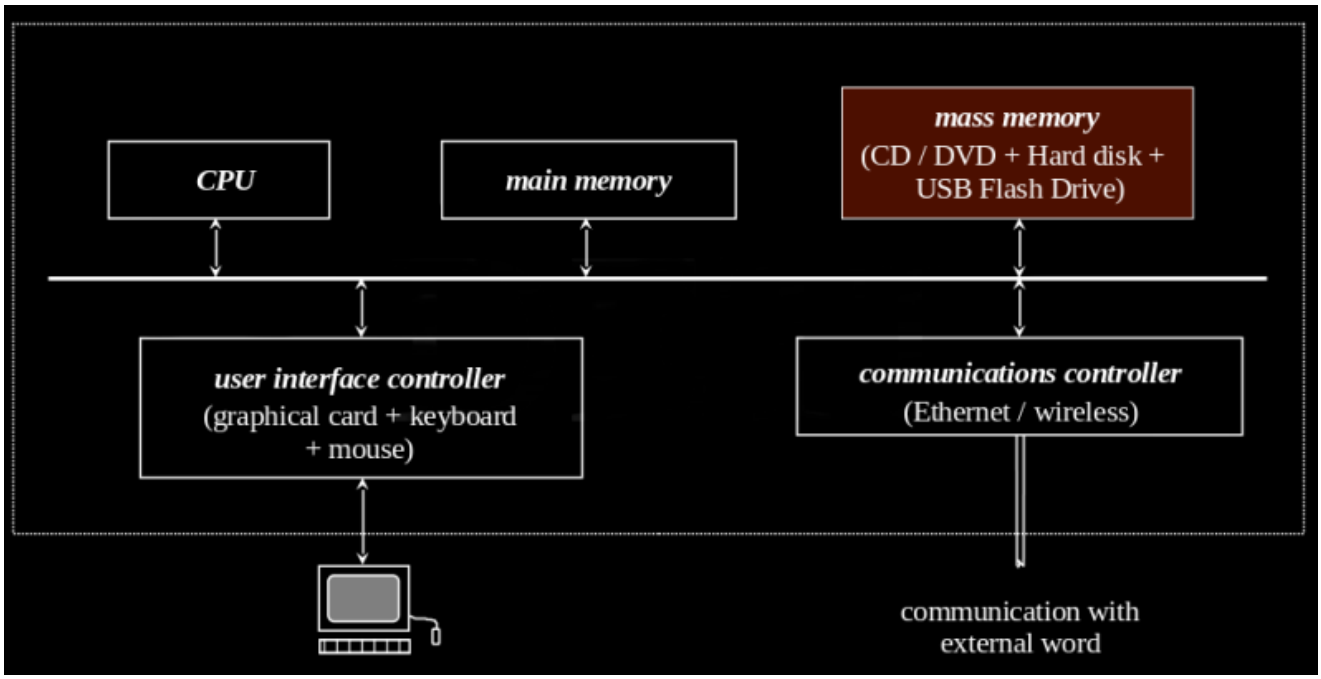
Duas possiveis estrategias: **demand paging** e **prepaging**

Demand paging - colocar nenhum e esperar por page faults - ineficiente.
Prepaging - colocar esses mais likely para serem referenciados. Primeira vez, as 2 pages mencionadas antes (code e stack). Proximas vezes, as que estavam em main memory quando o processo estava suspendido - mais eficiente.

File systems

Overview

The mass storage



Importancia da mass storage (secondary memory)

Storage do sistema operativo - Quando um sistema é ligado, existe apenas um unico programa na main memory (numa regiao ROM-like), o **boot loader**, cuja funçao principal é ler um programa maior de uma regiao especifica de mass storage. Esse programa é loaded para a main memory e corre o programa que implementa o ambiente de user interaction.

Warehouse of applications - Para um pc fazer cenas uteis, deve existir um lugar permanente onde se armazena as diferentes aplicações.

Warehouse of user files - Ainda mais, quase todos os programas durante a sua execução produzem, consultam e/ou mudam informação que devem ser permanentemente armazenados.

Propriedades de mass storage

- **non-volatility**
- **grande capacidade de armazenamento** - a informação manipulada pelos processos pode exceder a que está diretamente armazenada na suas proprias address spaces
- **acessibilidade** - Acesso a informação armazenada deve ser feito da forma mais simples e rapida possivel
- **integridade** - a informaçao armazenada deve ser protegida contra corrupção acidental ou nao.
- **partilha de informaçao** - a informação deve ser acessivel concorrentemente para multiplos processos que precisem.

File system é a parte do sistema operativo responsavel pela gerencia de acesso a conteudos na mass storage.

Mass storage

Tipos de devices de mass storage

Type	Technology	Capacity (GB)	Type of use	Transfer rate (MB/s)
CD-ROM	mechanical / optical	0.7	read	0.5
DVD	mechanical / optical	4-8	read	0.7
HDD	mechanical / magnetical	250-4000	read / write	480
USB FLASH	semiconductor	2-256	read / write	60(r) / 30(w)
SSD	semiconductor	54-512	read / write	500

Abstraçao operacional de mass storage

Cada dispositivo é representado por um array de NTBK storage blocks, cada um consistindo de BKSZ bystes (BKSZ vai de 256 a 32K). Acesso a cada bloco para read ou write pode ser feito de forma random.

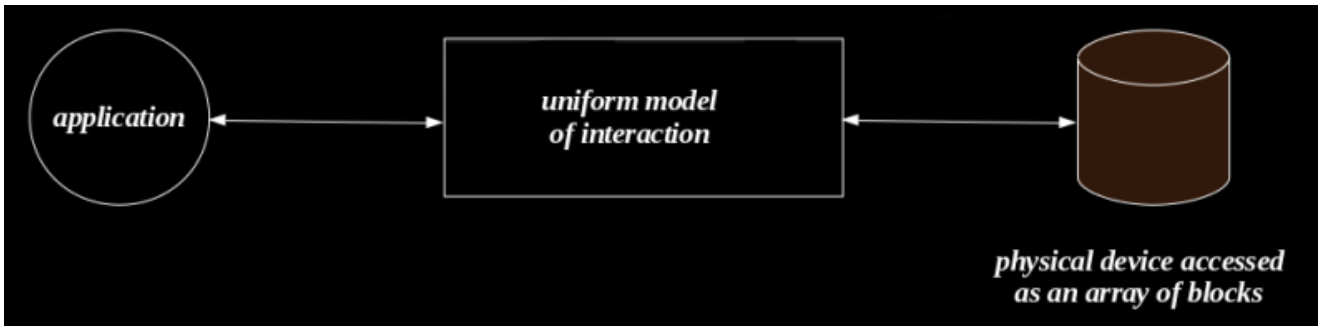
Isto chama-se **LBA - Logical Block Addressing** - Blocos sao localizados por um indice integer. ATA Standard incluia 22-bit, 28-bit e 48-bit LBA.

Um bloco é a unica unidade de interação. Logo, um unico byte **nao pode** ser acedido diretamente.

Apesar de criar um modelo uniforme, **LBA não é uma forma adequada** para uma aplicação aceder a dados na mass storage.

Manipulação direta da informação residente no physical device nao pode ser deixada inteiramente a responsabilidade do programmer.

É necessário um modelo uniforme de interação.



File concept

O que é um file

File é a unidade logica de storage em mass storage. Quer dizer que read e write é sempre feito na scope de um file,.

Elementos basicos de um file:

- **Identity name/path** - a forma generica de se referir a informaçao
- **Identity card** - meta-data (owner, size, permissions, times, ...)
- **Contents** - A propria informação, organizada como uma sequencia de bits, bytes, linhas ou registers, cujo **formato** é definido pelo criador do file e que tem de ser conhecido por quem o aceder.

Da perspetiva de um programmer, um file é um tipo de dados abstrato, caracterizado por um conjunto de atributos e operações.

Tipos de files

Do ponto de vista do sistema operativo, ha diferentes tipos de files:

- **Normais** - File cujo conteudo é da responsabilidade do user. Do ponto de vista do OS, é uma sequencia de bytes. e.g texto, imagens, video, aplicações...
- **Directory** - File usado para dar track, organizar e localizar outros files e directories.
- **Shortcut (symbolic link)** - file que contem uma referencia para outro file (de qualquer tipo) na forma de um path absoluto ou relativo.
- **Character device** - file que representa um device handled in bytes.
- **Block device** - file que representa um device handled in blocks.
- **socket** - file usado para inter-process e inter-machine communication.
- **(named) pipe** - file usado para inter-process communication.

Atributos de files

Atributos comuns de um file:

- **type** - um dos tipos referidos acima.
- **name/path** - a forma que users se referem ao file
- **internal identification** - a forma de como o file é conhecido internamente.
- **size(s)** - size em bytes de informaçao; espaço ocupado no disco.
- **ownership** - a quem pertence o file.
- **permissions** - quem pode aceder ao file e como.
- **acesso e modification times** - quando o file foi acedido ou modificado.
- **location of data in disk** - conjunto ordenado de blocks/clusters do disco onde se encontram os conteudos do file.

Operaçoes em files

Existe reading, writing, opening(se diretorio)/executing, mudar ownership...

Directories

Conceito

Discos comuns podem conter milhares ou milhoes de files. Seria impratico ter todos estes files no mesmo nivel de acesso.

Directory é um meio para permitir o acesso a conteudos do disco de uma forma hierarquica.

Um directory pode ser visto como um container contendo files ou outros directories.

Um directory pode ser implementado como um array (de size variavel) de **directory entries**.

Cada directory entry é um **key-value pair** que diretamente ou indiretamente associa o nome de um file aos seus atributos.

Nome e path

A existencia de hierarquia de file faz o nome insuficiente para referenciar um file num disco. Diferentes files na hierarquia podem existir tendo o mesmo nome. Como aceder a um file com apenas o nome? Não é facil, se possivel.

A noção de path deve ser introduzida. Um path é uma sequencia de nomes onde todos menos o ultimo devem ser directory names ou shortcuts apontando para directories.

Um Path pode ser absoluto ou relativo. Um absoluto diz a localização de um file de um root point, e o relativo diz de um intermediate point.

Em Unix, the root point is the root of a single, global file hierarchy. Devices de storage diferentes estao montados algures nesta hierarquia.

Em Windows, ha um root point por storage device (A:, B:, ...)

File system

Função do operating system

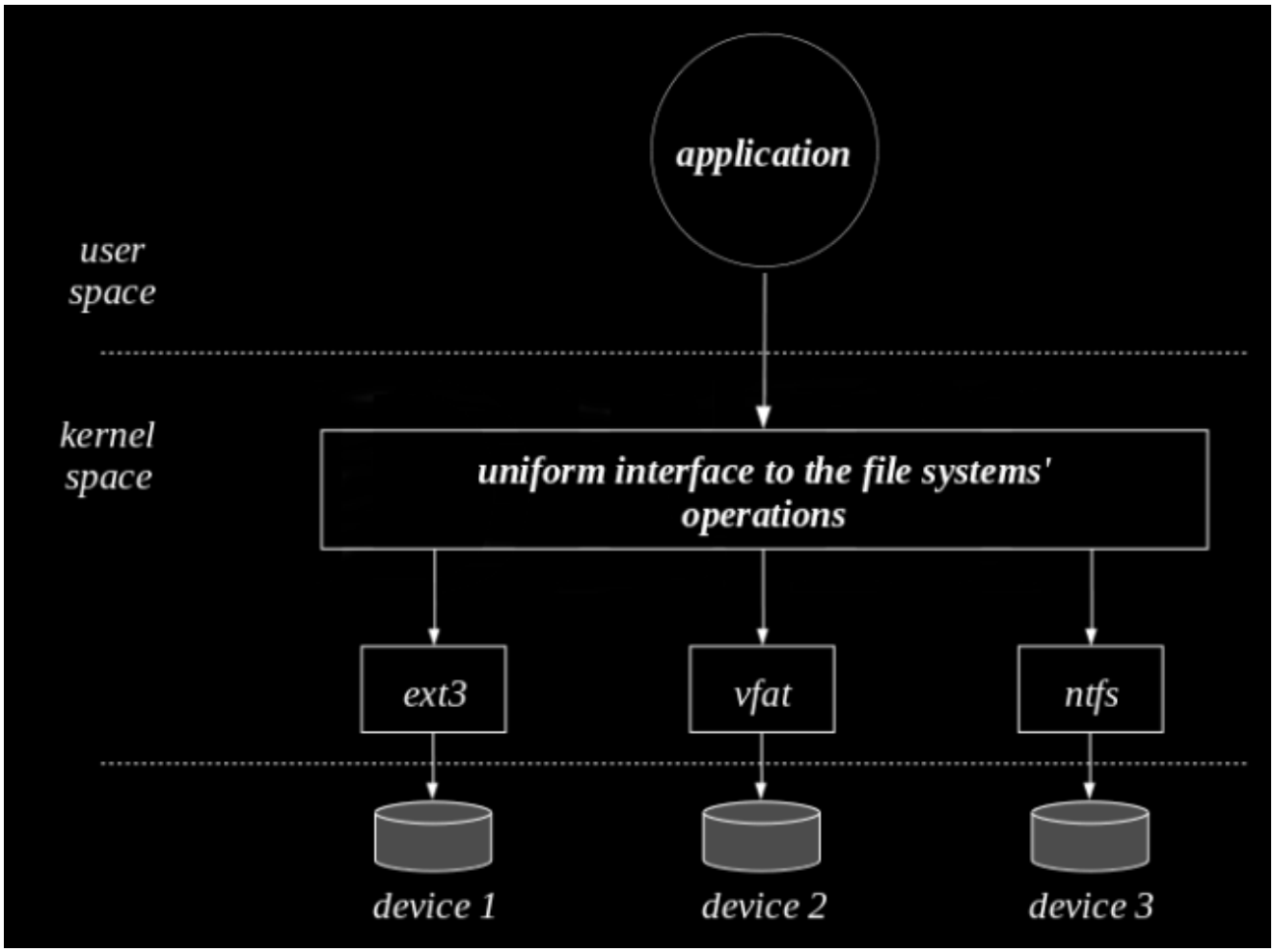
A funçao do operating system é implementar o conceito de file, fornencedo um conjunto de operaçoes (system calls) que estabelecem uma interface de comunicaçao simples e segura para acesso a conteudos da mass storage.

O file system é a parte do sistema operativo dedicada a esta tarefa.

Diferentes implementações do file data type levam a diferentes tipos de sistemas. Ex: ext3, FAT, NTFS, APFS, ISO 9660, ...

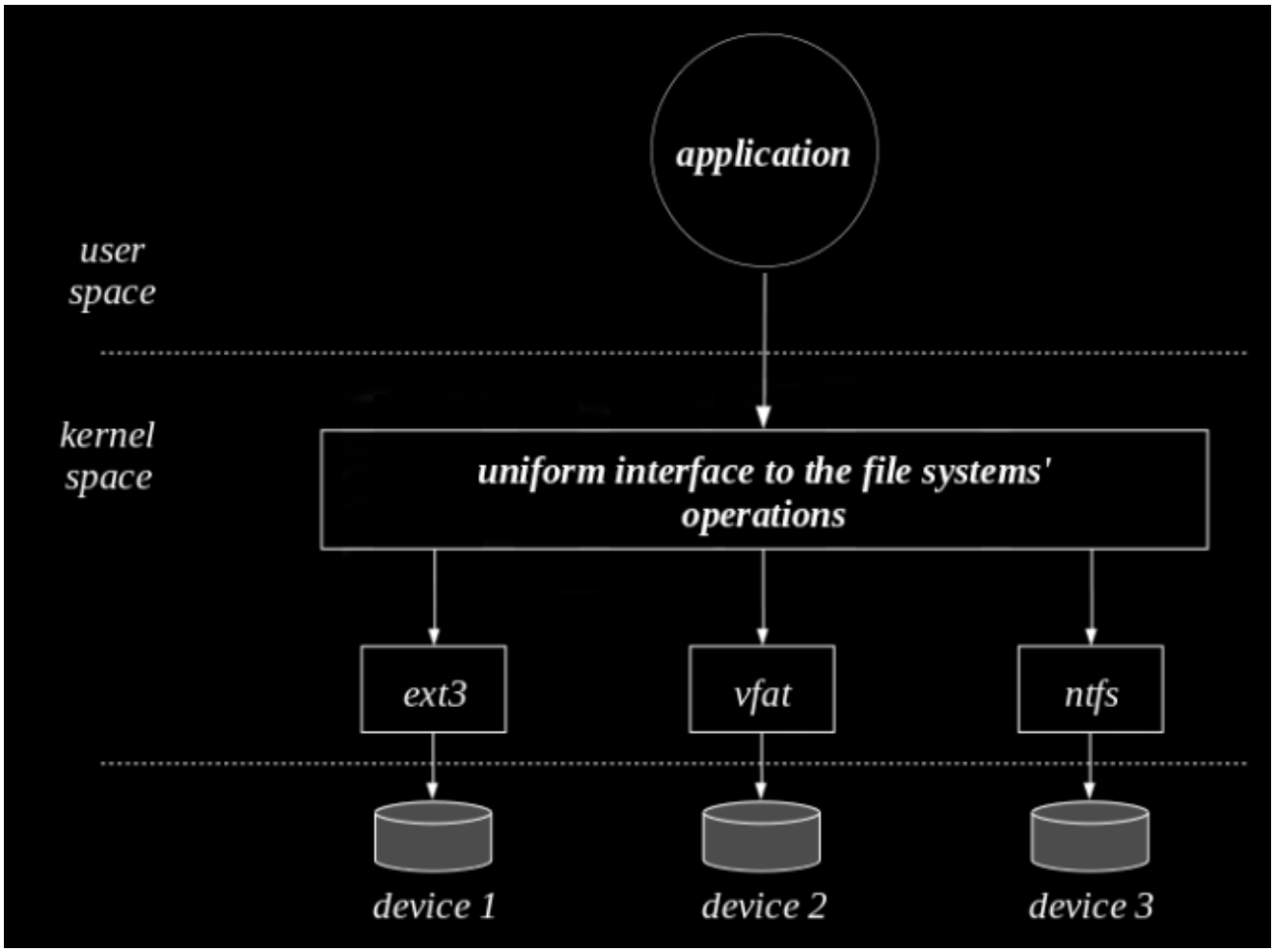
Hoje em dia, um unico sistema operativo implementa diferentes tipos de file systems, associados a diferentes physical devices, ou mesmo no mesmo. Esta feature facilita interoperability, estabelecendo um meio comum de partilha de informação entre sistemas heterogeneos.

Virtual file system



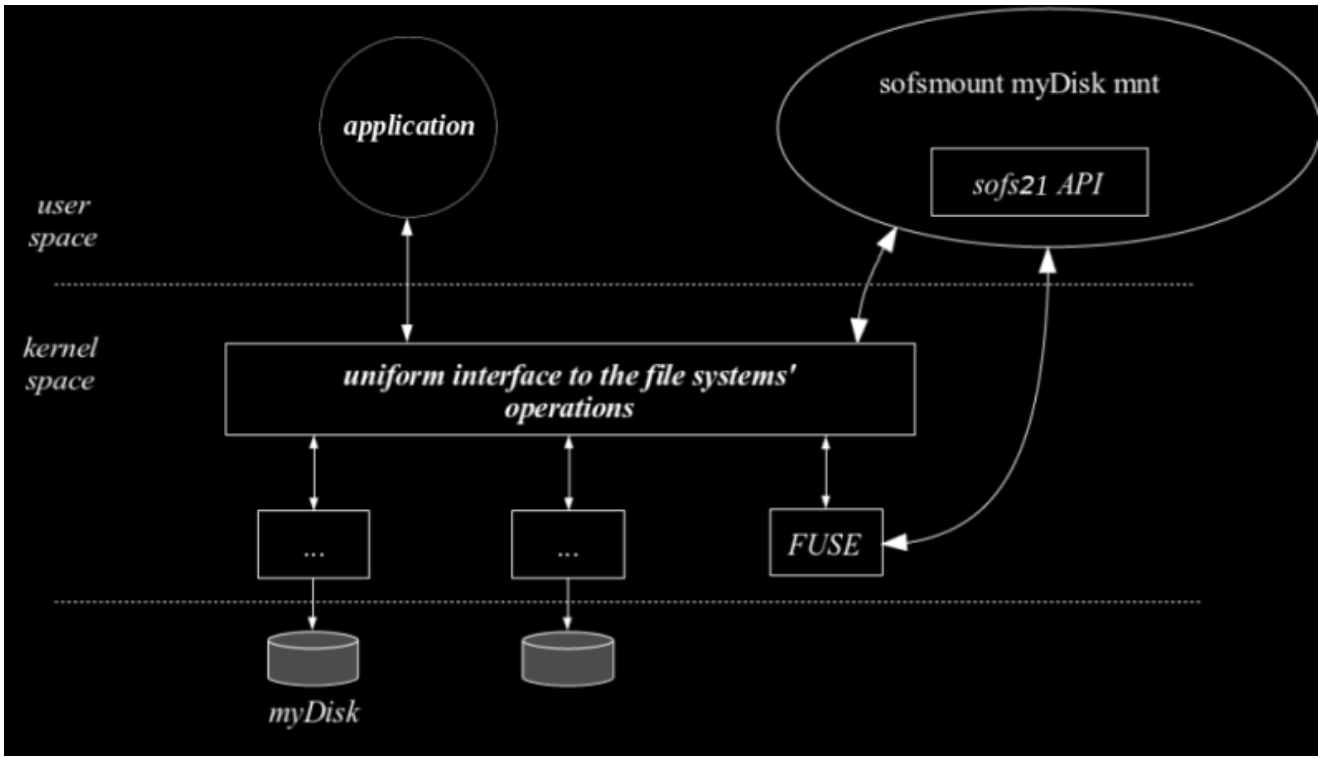
File system

File system como um modulo do kernel



Problema de segurança: correr em kernel space - Código malicioso ou errado pode danificar o sistema

File system as a FUSE module

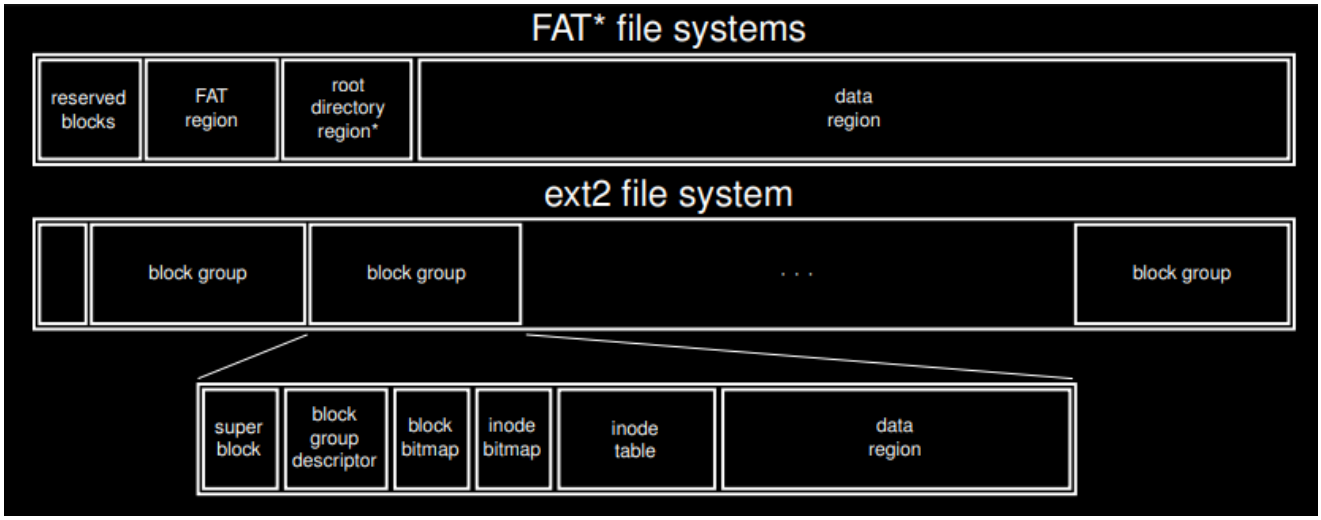


Seguro: correr em user space. Código malicioso ou errado apenas afeta o user.

Como implementar

Um problema de implementação é como organizar o storage space do device, visto como um array de blocos, para obter a vista abstrata desejada.

File systems diferentes estao relacionados a arquiteturas internas diferentes.



Data blocks

Alguns pontos

O bloco (cluster no Windows) é a unidade de alocação para conteudos de files. Um bloco pode ser um unico setor do disco ou uma sequencia contiguous de setores, normalmente em powers of 2.

Blocos nao se partilham entre files. Em geral, um bloco em uso pertence a um unico file.

O numero de blocos necessarios por um file para armazenar a informação é dada por:

Nb = roundup(size / BlockSize)

- Nb pode ser muito grande - se block size é 1024 bytes, um file de 2 GB precisa de 2M Blocks.
- Nb pode ser muito pequeno - um file de 0-bytes nao precisa de blocos para dados.

É impratico que todos os blocos usados por um file estejam contiguous num disco. O acesso a dados de um file é, em geral, nao sequencial, mas random instead.

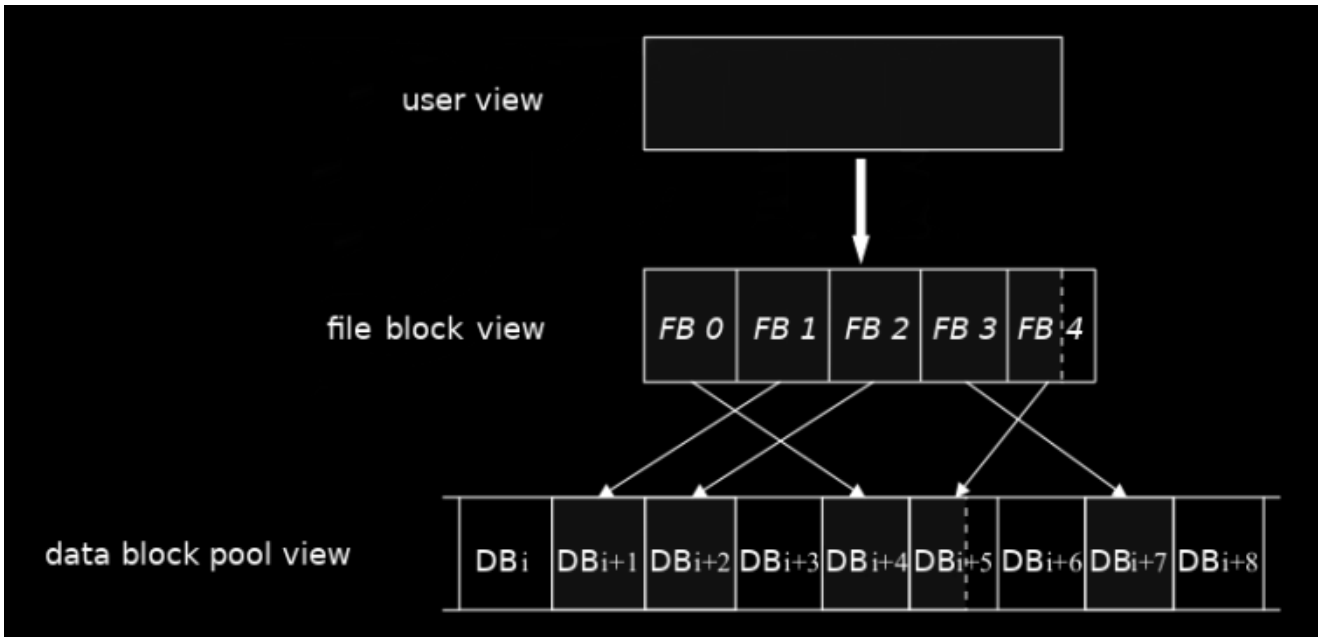
É necessária uma estrutura de dados flexivel, em size e location.

File content views

A vista do programador: Um file é um continuum de bytes

A vista do file block: Um file é uma sequencia de blocos

A vista do data block: em geral, um filo está espalhado pela data block region.

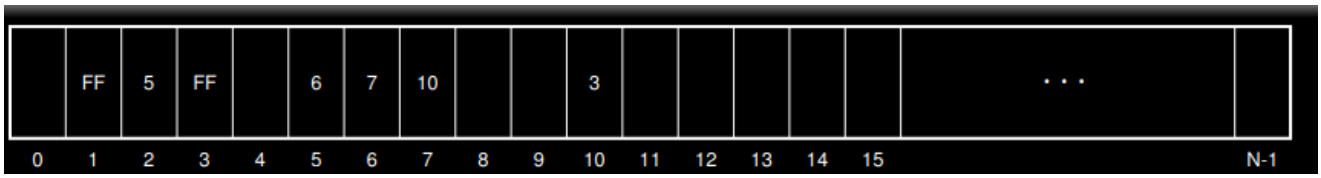


Sequencia de blocos num file: FAT system approach

Como é armazenada a sequencia de (referencias a) data blocks num sistema FAT.

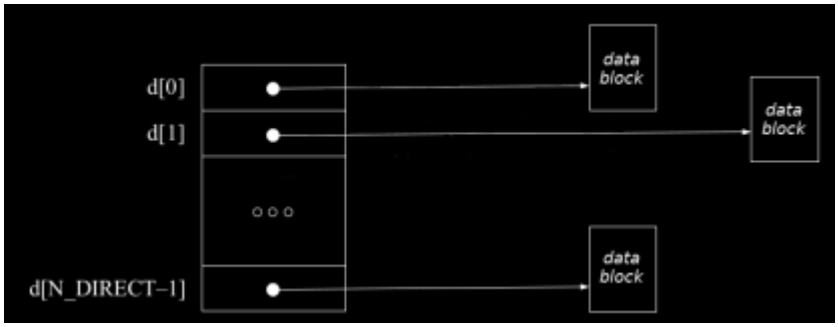
A primeira referencia é diretamente armazenada na entrada do directory.
De seguida, a **file allocation table (FAT)** permite identificar as block references que faltam. Esta table é um array de referencias, armazenadas numa parte fixa do disco. Cada entrada pode ter 12, 16, ou 32 bits.

Assumindo um sistema FAT16 e que a sequencia com os dados do file é 2,5,6,7,10,3, os conteudos da table sao do tipo:

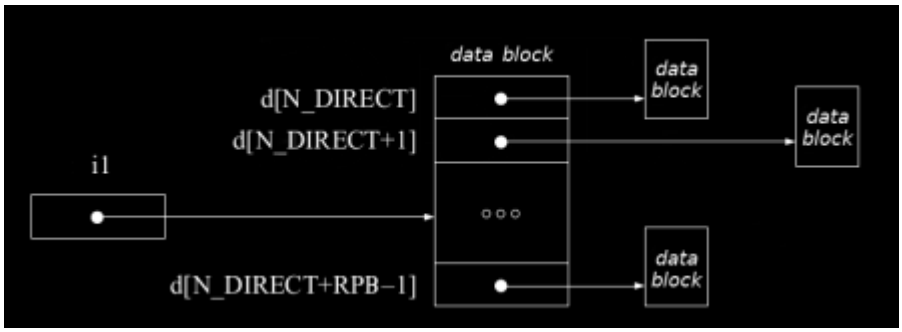


Sequencia de blocos num file: Approach do sofs21

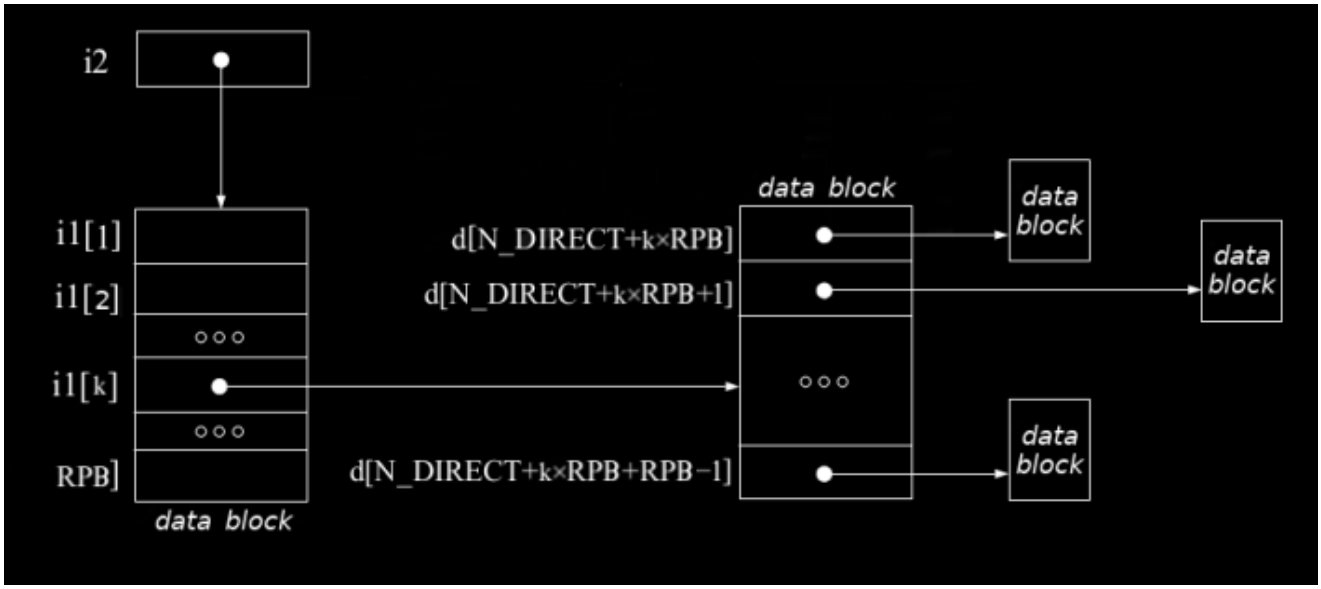
As primeiras referencias estao diremante armazenadas na **inode** do file. Uma inode é um record que contem a metadata de um file.



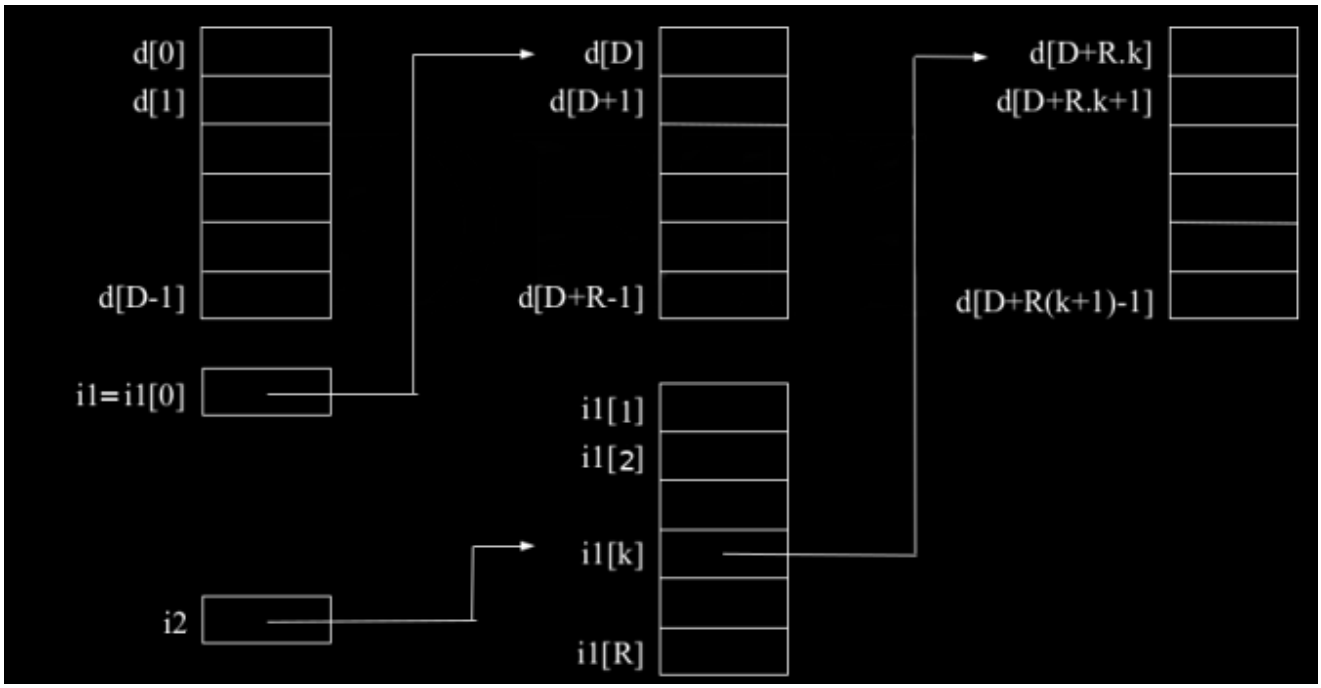
Depois, inode field `i1` aponta para um data block com referencias.



Finalmente, o field `i2` aponta para um data block que extende `i1`.



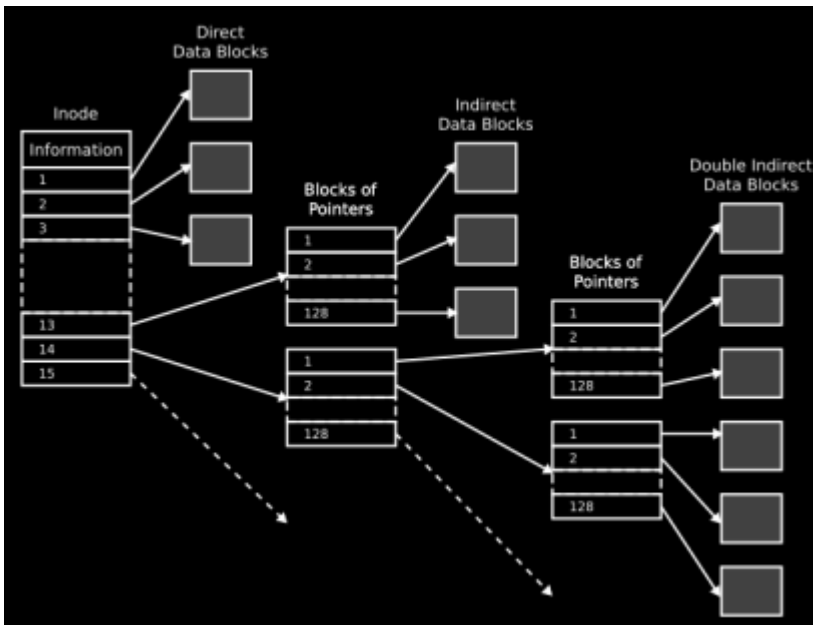
Metendo tudo junto:



Um file pode conter "**holes**", correspondentess a referencias null cobertas pelo size e representando streams de zeros.

Sequencia de blocos num file: Approach do ext2

A approach é a mesma de antes com um triple indirect pointer adicional. Ha 12 direct pointers, 1 indireito, 1 duplamente indireto e 1 triplamente indireto



Um file pode conter "**holes**", correspondentess a referencias null cobertas pelo size e representando streams de zeros.

Lista de data blocks livres

Temos que saber que blocos tao livres a um dado momento. Se um file cresce, requisitando um novo bloco, qual deve ser alocado?

No FAT:

- Uma FAT entry com valor 0 representa um free cluster.
- Alocar um novo block requer pesquisa na tabela FAT à procura de uma entrada com esse valor.

No ext2:

- Ha um secção num block group contendo um bitmap de data blocks livres/alocados no grupo.
- Alocar um novo data block requer pesquisar o bitmap à procura de um bit a 0.

Inodes

O que são

Em Unix, o inode (**identification node**) é muito importante.

Corresponde à identity card de um file e contem file type, owner info, file permissions, acess times, file size e sequencia de blocos com os conteudos do file.

O nome/path nao estao no inode. Tao na entrada da directory

Num sistema ext2, em cada block group, ha uma regiao reservada para inodes, a **inode table**

Tambem ha um *inode bitmap* que representa os inodes livres/alocados.

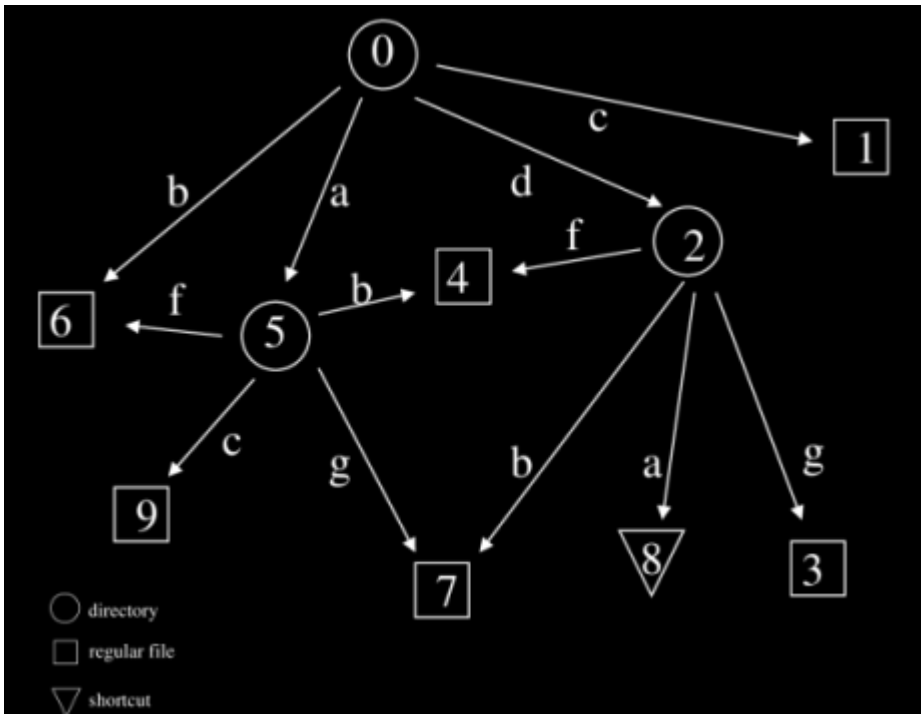
Hierarquia de files em Unix file systems

Cada file usa apenas um inode

Mesmos inodes podem ter diferentes pathnames

Hierarquia de ficheiros pode nao ser uma arvore.

Os conteudos de um disco podem ser vistos como um grafo, onde nodes sao os files, tendo cada um o seu inode associado



Implementaçao de directory

FAT

Uma directory é um conjunto de directory entries, sendo estas key-value pairs que diretamente associam file names com file attributes.

Todas as entradas sao de 32 bytes.
Normal name field é composto de 8 (name) + 3 (extension) carateres.
Nao existe ownership field
Um dos fields indica o primeiro cluster.

ext2

Uma directory é um conjunto de directory entries, sendo estas key-value pairs que diretamente associam file names com inodes.

Tamanho das entradas depende na name length. Nao existe a noçao de extensao.

Entradas de directories diferentes podem apontar para o mesmo inode - **hard links**.

Os atributos do file estao no inode.

Input / Output

Funçao do operating system

Dois pontos de vista diferentes sao normalmente considerados para a funçao do operating system em gestao de I/O devices.

- **perspetiva do user** - fornece ao application programmer uma API que é conceptualmente simples, uniforme e independente do device especifico.
- **perspetiva do sistema** - isolating the different devices from direct access by user processes by introducing a functional layer that directly controls the devices - mandar comandos, transferir dados, handle interrupts ...

Interface com o processador

Há 2 diferentes componentes a considerar:

- **O proprio device** - sistema fisico que armazena informaçao e a converte para uma forma acessivel do exterior.
- **Device controller** - circuito eletronico, parte do computador, que funciona como uma interface com o device.

Do **ponto de vista do OS**, o controller é o unico componente relevante.

Hoje em dia, controllers sao muito versateis, minimizando a funçao do OS na gestao.

Tipos de devices

Em termos de transferencia de informação, I/O devices caem em 2 categorias.

- **character-type devices** - a tranferencia de informaçao é baseada numa stream de bytes, de tamanho variavel.
- **block-type devices** - a transferencia de informação é baseada numa constante e numero pre definido de bytes, o **block**, tipicamente com um valor igual a um power of 2 entre 512 Bytes e 16 KB.

A forma que a transferencia é feita depende do bus usado.

- bytes (8 bits), 2-bytes (16 bits), 4-bytes (32 bits) or 8-bytes (64 bits).

A rate de transferencia depende no tipo de device.
Pode variar de dezenas de bytes (e.g teclado) para milhares de megabytes (SATA

ou disco USB3).

Device controller

Um **generic controller**, de um ponto de vista de programação, pode ser visto como um conjunto de registers.

- **control registers** - com varias funçoes:
  - Configurar o defice
  - Definir o tipo de interaçao com o processador (polled I/O, interrupt-driver I/O, DMA-based I/O).
  - Em controllers complexos, executar um comando.
- **status register** - representa o estado interno do device para:
  - Indicar o sucesso da ultima oepraçao
  - Indicar o falhanço e erros da ultima oepraçao
  - Indicar se ta pronto para receber um novo comando.
- **data registers** - usado para a comunicação.
  - Valores escritos no register **out** sao mandados para o device.
  - Valores lidos do register **in** vieram do device.

Em **character-type** devices, os comandos write e read sao implicitos. Um valor escrito no **out** é mandado para o device e um valor recebido do device é metido no **in**.

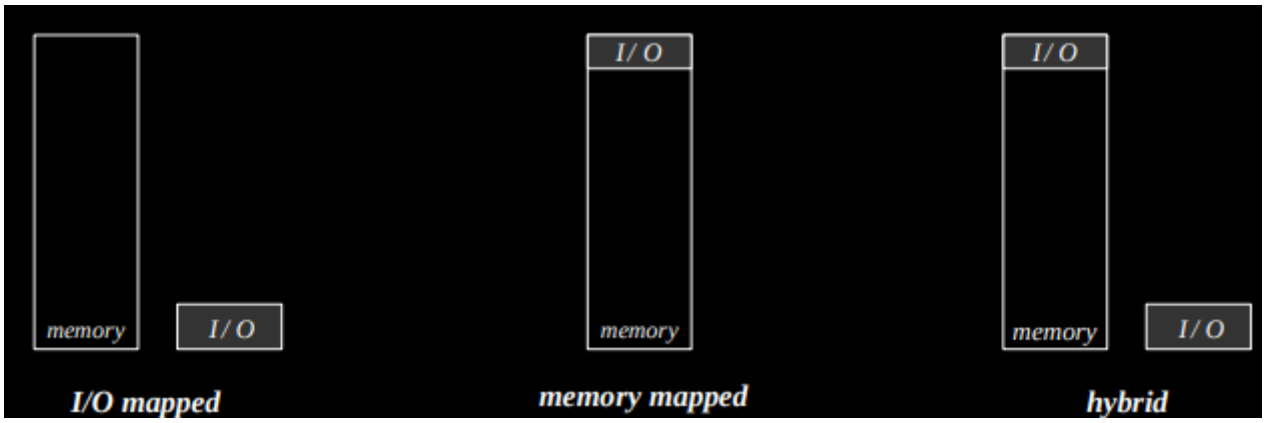
Em **block-type**, a transferencia começa baseada num comando explicito. A data register é em geral unica, **in-out**, e a direçao da transferencia depende do comando dado.

I/O address modes

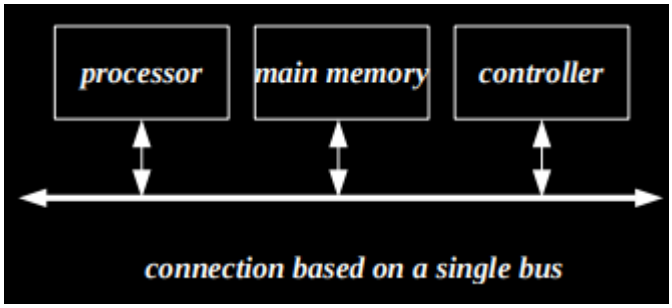
Há 3 diferentes formas possiveis para aceder aos registos internos de um controller

- **I/O mapped** - controllers sao mapeados numa especifica I/O address space e os registers sao acedidos por instruções especificas (*in* e *out*).
- **memory-mapped I/O** - controllers sao mapeados na memory address space e os registers sao acedidos por memory access instructions (*load* e *store*).
- **hybrid** - controllers sao mapeados numa address space I/O especifica, MAS data buffers sao mapeados na memory address space para facilitar

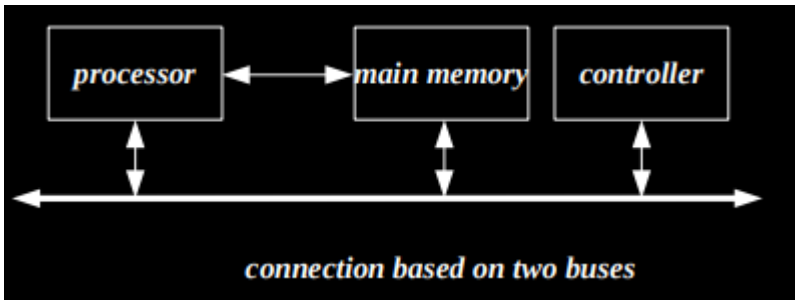
comunicação.



Numa configuraçao simples, todos os recursos de um computador sao conectados usando um unico bus.



Hoje em dia, pcs usam um broadband bus dedicado à transferencia de dados entre o processador e a main memory, para se aproveitar da clock frequency.



Nestes casos, a memoria é *dual-port* para permitir concorrencia com a transferencia de dados de devices DMA-based.

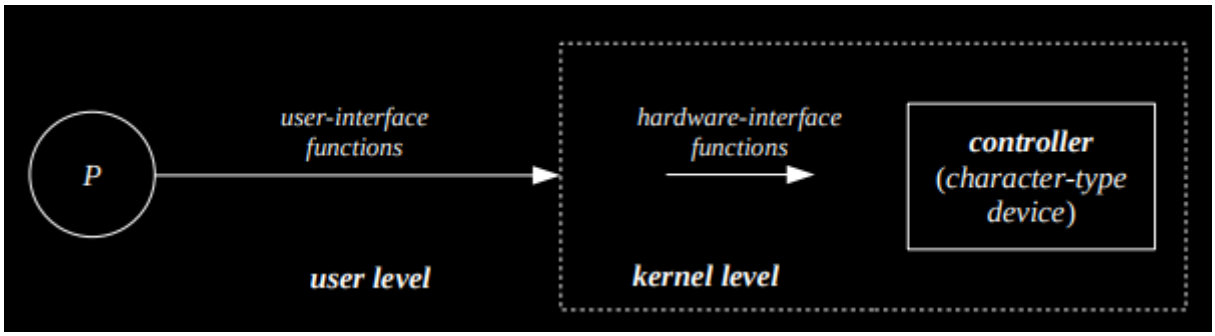
Nem todos os controllers estao conectados ao mesmo bus.

Objetivos de I/O programming

O ambiente fornecido pelo sistema operativo para a comunicaçao com I/O devices deve:

- **ser independente das especificaçoes do device** - devices devem ser vistos como genericos.
- **suportar um naming mechanism uniforme** - device names devem consistir de strings de caraters sem qualquer sentido particular.
- **decouple devices from user processes** - a vasta maioria de devices I/O trabalham de uma forma assincrona - transferencias de dados para main memory sao triggered por interrupts. Da perspetiva do user, é mais facil fazer comunicaçoes de forma sincrona - o processo bloqueia ate que condições estejam validas para comunicação acontecer.
- **gerir acesso a preemptable e non-preemptable devices de forma uniforme** - comunicação com preemptable devices pode ser partilhada por multiplos users uniformemente. Comunicação com non-preemptable devices deve acontecer em mutual exclusion, ou num regime dedicado. O sistema operativo tem que identificar as diferentes situaçoes e assegurar coordenaçao correta.
- **fazer error management de forma integrada** - a deteçao de erros deve ser feita as close to the device as possible para permitir a sua possivel recovery de forma transparente. A politica geral deve ser para apenas reportar o erro à upper layer se a lower layer nao aguentar.

Polled I/O

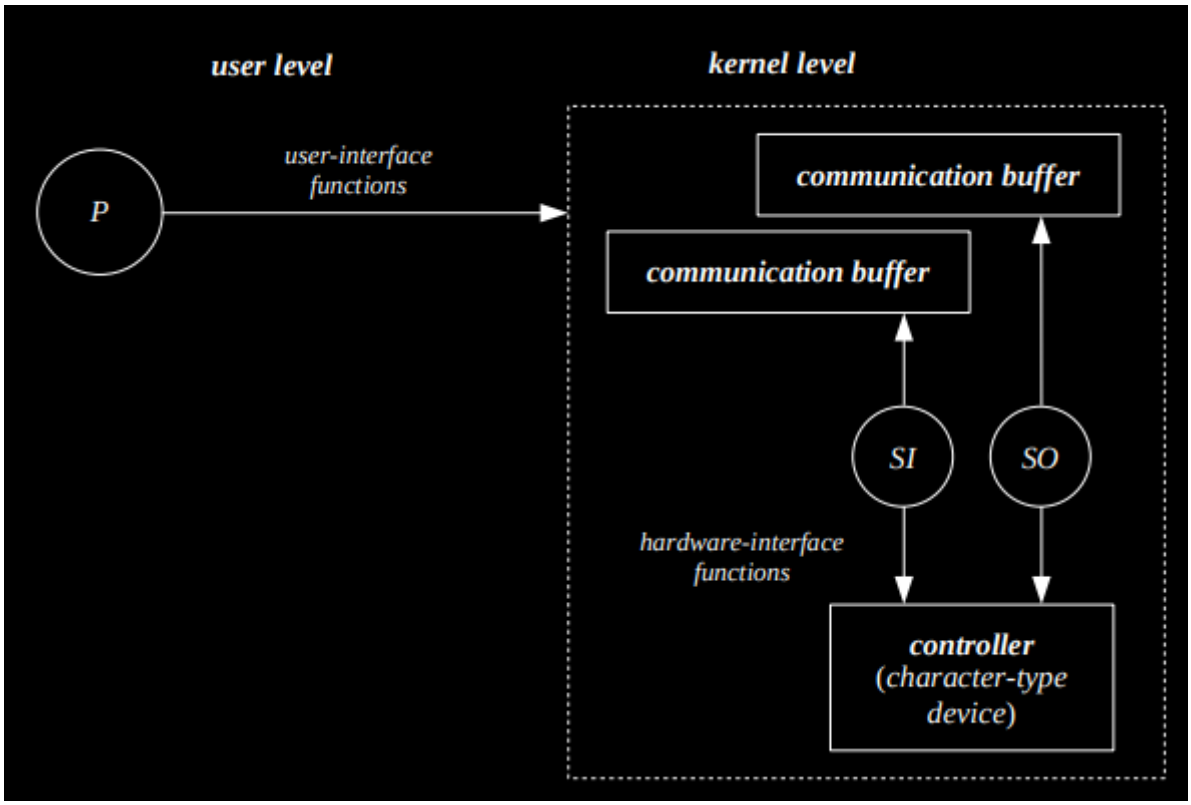


Em polled-I/O approach, nao ha decoupling. É o user process diretamente responsavel por comunicaçao. Rotinas de comunicaçao de device sao system calls que diretamente implementam hardware access.

A soluçao mais simples, mas pouco eficiente, porque o processador fica busy waiting.

Interrupt driven I/O

Nesta approach, acesso ao device é feito por dois processos de sistema, **SI** e **SO**, triggered por interrupçao. Comunicação entre o user process e estes processos de sistema é feita por dois buffers de comunicaçao.

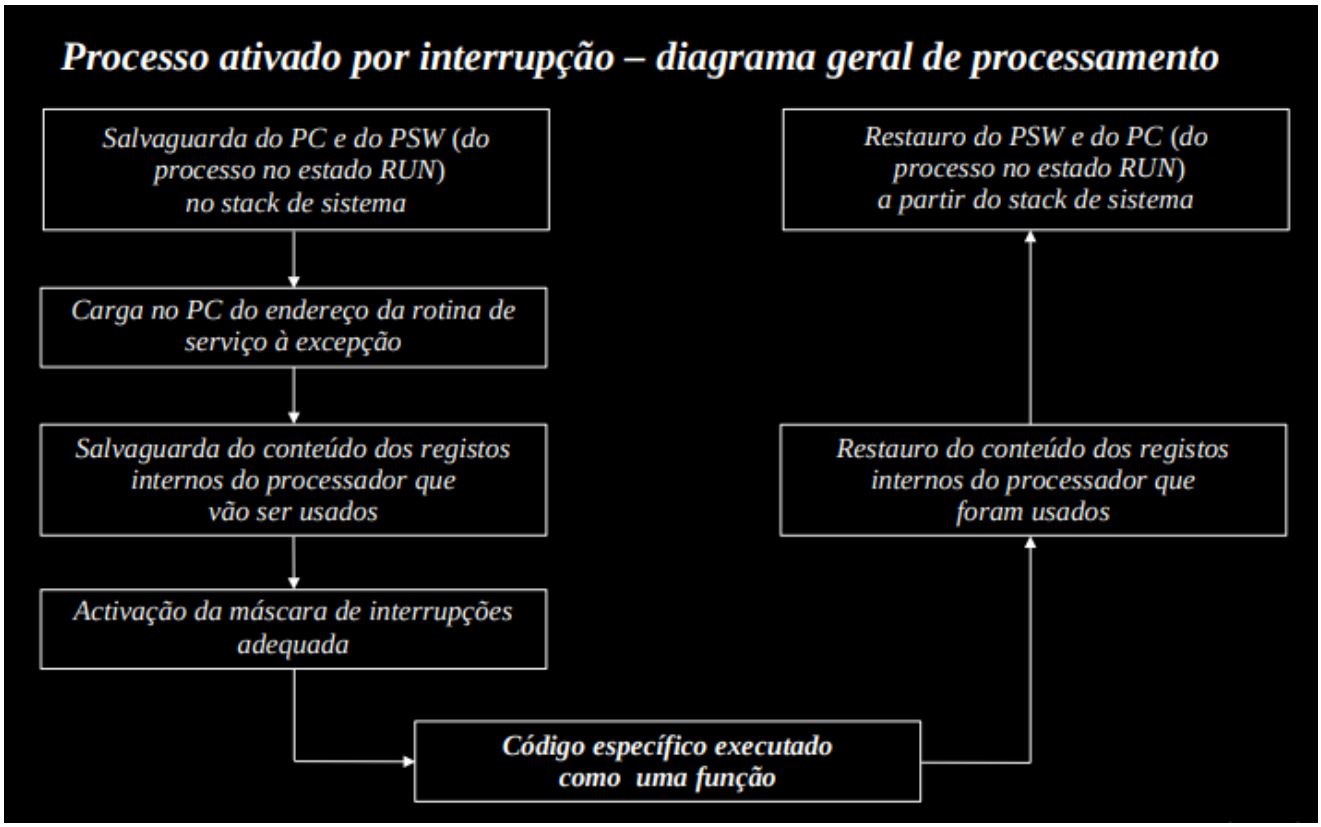


Os processos **SI** e **SO** representam rotinas de serviço para interrupts triggered pelo controller, quando, respetivamente, a input daata register contem um novo byte, ou a output data register ta vazia e pronta para receber um novo byte. Eles sao processos especiais que correm ate exaustao. Eles apenas perdem controlo do processador se outro processo de maior prioridade fica scheduled.

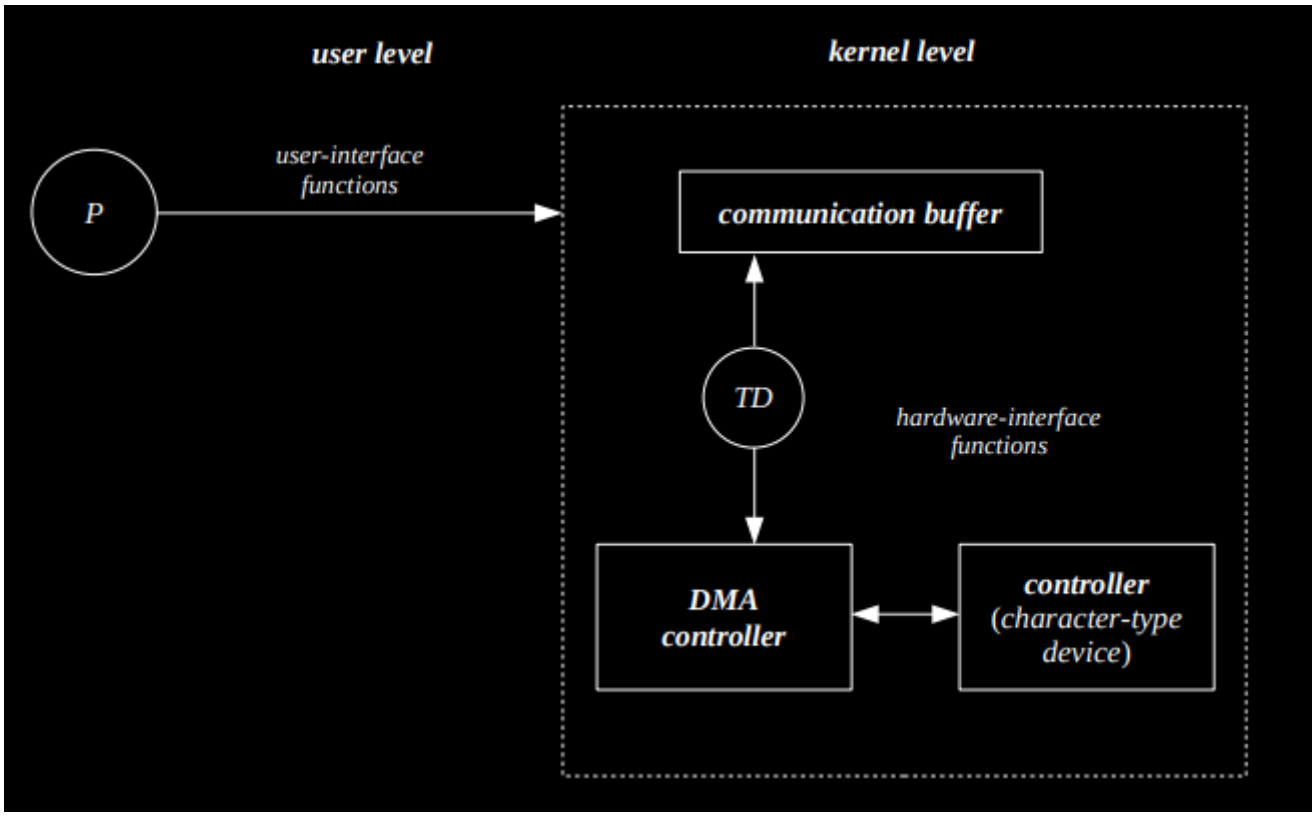
As funçoes de user interface implementam acesso a buffers de comunicaçao decoupling the device from the user process.

Uma soluçao mais complexa, mas mais eficiente, porque o user process bloqueia, libertando o processador se condiçoes para transferir dados nao sao validas.

Diagrama geral de processamento



DMA-based I/O



Nesta approach, o DMA controller está diretamente conectado ao device controller, ou faz parte dele. O principio underlying é que quando o device controller quer transferir dados, ativa um request transfer input no controller DMA.

Como resultado, toma controlo do bus e faz uma das duas operaçoes:

- Ler um byte ou word do device controller data register e depois escrever esse valor para memory (data input).
- Ler um byte ou word de memoria e depois escrever esse valor para o device controller data register (data output).

Teste

Há sempre 3 grupos sobre os mesmos temas: Scheduler, Gestao de Memoria e Concorrencia.

- Para o grupo de scheduler é preciso saber:
  - politicas round robin, fcfs e spn.
  - Saber desenhar a politica escolhida no grafico.
  - Saber os conceitos de swap in/out (quando precisamos de meter processos na main memory e ta cheia, damos swap out de processos READY ou BLOCKED, i.e metemos esses processos na swap area para arranjar espaço).
  - Saber o grafico long term/short term/medium term e todas as transiçoes.
  - Sistema preemptive e non/preemptive
  - Sistema interativo (é preciso preemption (frequente) para o user ter uma exp agradavel a usar o sistema.)
  - Turnaround time e todos esses conceitos acho que nao é preciso saber porque no teste dizem o que é. Vao ter que calcular e.g turnaround time no grafico que desenharam na politica pedida.
  - Distinguir multiprocessing de multiprogramming.
  - Distinguir Cpu burst de I/O burst.
- Gestao de memoria
  - Vao escolher uma MMU qualquer (contigous allocation, paging, seg, block) e vao pedir para explicar como funciona o logical address translation to physical address.
  - Nessa MMU tambem é preciso saber quando os respetivos registers trocam de valor (em que fases do scheduler e como/porquê).

    - Saber Politicas basicas de page substitution: LRU, NRU e FIFO. Saber por que NRU é melhor que LRU e saber por que LRU é ineficiente.
    - Saber Politicas de allocation: next fit, best, worst, next.

- Concorrencia

    - Vao dar um pedaço de codigo com forks/waits e podem conter uso de semaforos/threads.
    - Sequencias possiveis no pedaço de codigo.
    - Saber conceitos: race condition/starvation/deadlock/mutual exclusion.
    - Saber como funcionam forks/waits para poder dizer que codigo é executado por quem.
    - Comparar wait/signal com up/down. (Resposta)
    - Rewrite codigo adicionando semaforos/threads (o stor é que escolhe) para nao haver race conds.

- Random Grupo

- O 4o grupo é sobre um assunto random mas provavelmente sobre storage

- Pode sair comunicaçoes entre processos, deadlock avoidance/prevention/detection, storage, I/O comm etc.

Ja saiu sobre FAT16, sofs2016 e banker's algorithm (deadlock avoidance).

Wait-signal relacionado com up-down

- Se o *signal* é emitido e nao ha nenhuma thread em *wait*, o signal perde-se. Logo, o *wait* tem que correr primeiro necessariamente. Se um *up* ocorrer, o semaforo fica levantado à espera que apareça o *down*.
- O *wait* fica sempre à espera de um futuro signal. Qualquer *signal* anterior nao acorda o *wait*. O *down* pode se acordar com um *up* que acordou previamente.
- O *wait* bloqueia logo no codigo. O down so bloqueia se o semaforo ja for == 0. Se o semaforo for diferente de 0, nem vai bloquear sequer, nao ocorrendo sincronizaçao. A sincronização é importante em ambientes multithreading pois todos os processos partilham o mesmo espaço de endereçamento.