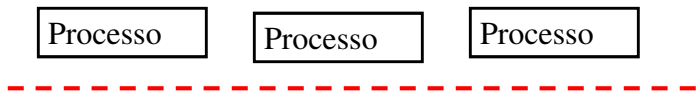
Núcleo, Processos e Tarefas

Sistemas Operativos

2011 / 2012

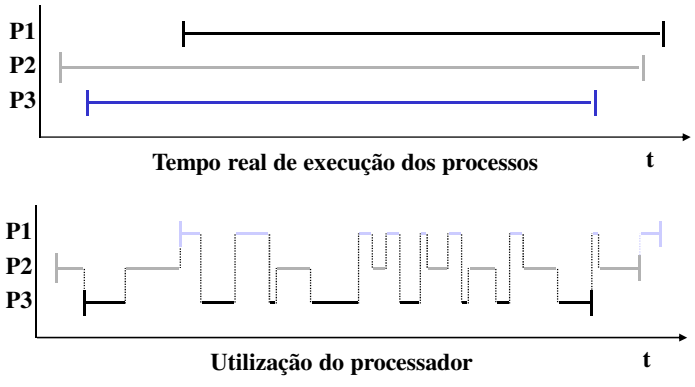
1. Processos



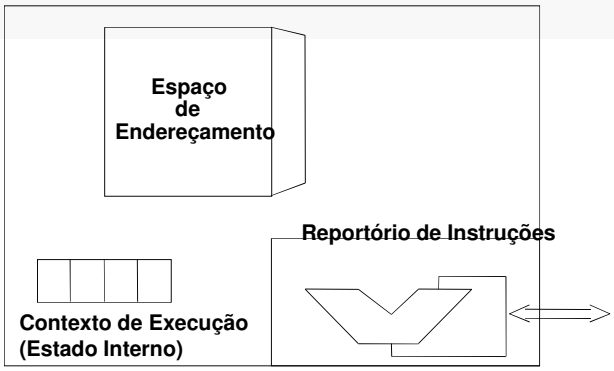
Multiprogramação

- Execução, em paralelo, de múltiplos programas na mesma máquina
- Cada instância de um programa em execução denomina-se um **processo**
- **Pseudoparalelismo** ou **pseudoconcorrência** – implementação de sistemas multiprogramados sobre um computador com um único processador
  - Considerando um grau de tempo fino, o paralelismo não é real

Pseudoconcorrência

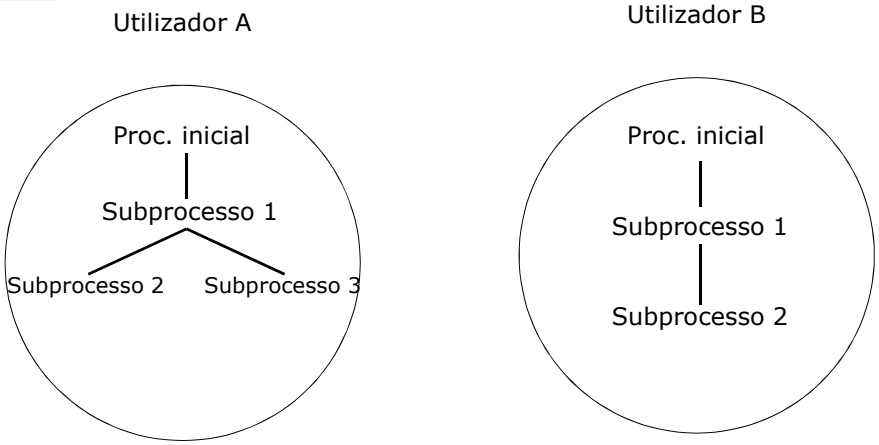


Processo Como Uma Máquina Virtual



- Elementos principais da máquina virtual que o SO disponibiliza aos processos

Hierarquia de Processos



- Certas informações são herdadas

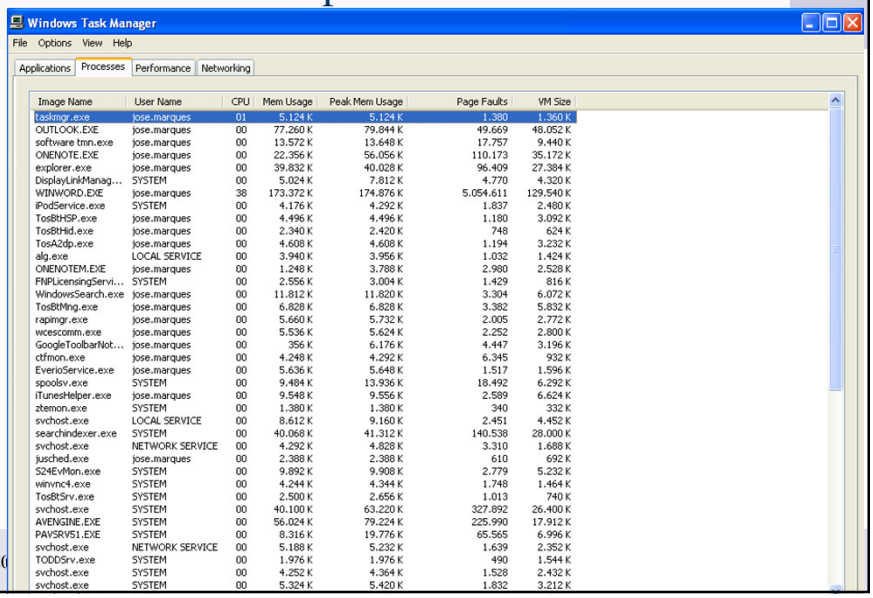
Modelo: Objecto "Processo"

- Propriedades
  - Identificador
  - Programa
  - Espaço de Endereçamento
  - Prioridade
  - Processo pai
  - Canais de Entrada Saída, Ficheiros,
  - Quotas de utilização de recursos
  - Contexto de Segurança
- Operações – Funções sistema que actuam sobre os processos
  - Criar
  - Eliminar
  - Esperar pela terminação de subprocesso

Exemplo: Unix

```
ps -ef | more
    UID   PID  PPID  C    STIME TTY  TIME CMD
   root     0     0  0   Sep 18 ?    0:17 sched
   root     1     0  0   Sep 18 ?    0:54 /etc/init -
   root     2     0  0   Sep 18 ?    0:00 pageout
   root     3     0  0   Sep 18 ?    6:15 fsflush
   root   418     1  0   Sep 18 ?    0:00 /usr/lib/saf/sac -t 300
daemon   156     1  0   Sep 18 ?    0:00 /usr/lib/nfs/statd
```

Exemplo: Windows



Processos em Unix

(Sob o ponto de vista do utilizador)

Processos em Unix

- Identificação de um processo
  - um inteiro designado por PID
  - Alguns identificadores estão pré atribuídos:
    - processo 0 é o *swapper* (gestão de memória)
    - o processo 1 *init* é o de inicialização do sistema
- Os processos relacionam-se de forma hierárquica
  - O processo herda todo o ambiente do processo pai
  - O processo sabe quem é o processo de que descende
  - Quando o processo pai termina os subprocessos são adoptados pelo processo de inicialização (pid = 1)
    - Continuando a executar-se
- Os processos têm prioridades variáveis.
  - Veremos as regras de escalonamento mais adiante.

Criação de um Processo

```
id = fork()
```

A função não tem parâmetros!

O contexto do processo pai é copiado para o filho, incluindo:

- **Código**
- **Cópia inicial dos dados e pilha**

A função retorna o PID do processo.

Este parâmetro assume valores diferentes consoante o processo em que se efectua o retorno:

- ao processo pai é devolvido o "pid" do filho
- ao processo filho é devolvido 0
- -1 em caso de erro

Retorno de uma função com valores diferentes → não existente na programação sequencial

Exemplo de fork

```
main() {
    int pid;

    pid = fork();
    if (pid == 0) {
        /* código do processo filho */
    } else {
        /* código do processo pai */
    }

    /* instruções seguintes */
}
```


Terminação do Processo

- Termina o processo, liberta todos os recursos detidos pelo processo, ex.: os ficheiros abertos
- Assinala ao processo pai a terminação

```
void exit (int status)
```

Status é um parâmetro que permite passar ao processo pai o estado em que o processo terminou.

Normalmente um valor negativo indica um erro

Terminação do Processo

- Em Unix existe uma função para o processo pai se sincronizar com a terminação de um processo filho
- Bloqueia o processo pai até que um dos filhos termine

```
int wait (int *status)
```

Retorna o pid do processo terminado. O processo pai pode ter vários filhos sendo desbloqueado quando um terminar

Devolve o estado de terminação do processo filho que foi atribuído no parâmetro da função exit

Exemplo de Sincronização entre o Processo Pai e o Processo Filho

```
main () {
    int pid, estado;

    pid = fork ();
    if (pid == 0) {
        /* algoritmo do processo filho  */
        exit(0);
    } else {
        /* o processo pai bloqueia-se à espera da
           terminação do processo filho  */
        pid = wait (&estado);
        printf("Resultados: ……");
    }
}
```

Execução de um Programa

```
int execl(char* ficheiro, arg0, argl,…, argn,0)

int execv(char* ficheiro, *argv [])
```

Caminho de acesso ao ficheiro executável

Por convenção o arg0 é o nome do programa

Argumentos para o novo programa. Podem ser passado como apontadores individuais ou como um array de apontadores. Estas parâmetros são passados para a função main do novo programa e acessíveis através do argv

Exercício: Shell simples

- Como programar a seguinte *shell* simplificada?
  - Espera que utilizador digite nome do ficheiro com programa a executar
  - Quando isso acontece, processo da shell cria processo filho que executa o ficheiro indicado
  - Processo da shell bloqueia-se até que o processo filho termine
  - Quando desbloqueado, processo da shell volta a esperar por nova ordem do utilizador

Exercício: Shell simples

```
while (TRUE){
    prompt();
    read_command (command, params);









}
```

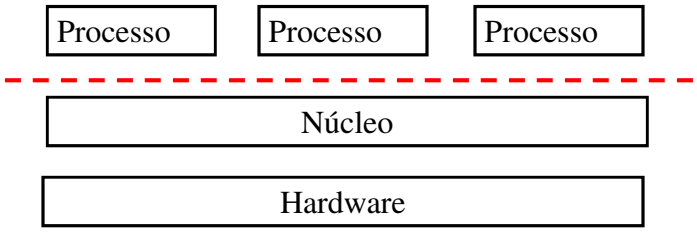

Exercício: Shell simples

```
while (TRUE){
    prompt();
    read_command (command, params);

    pid = fork ();
    if (pid == 0) {

    } else if (pid > 0) {

    }
    else {

    }
}
```

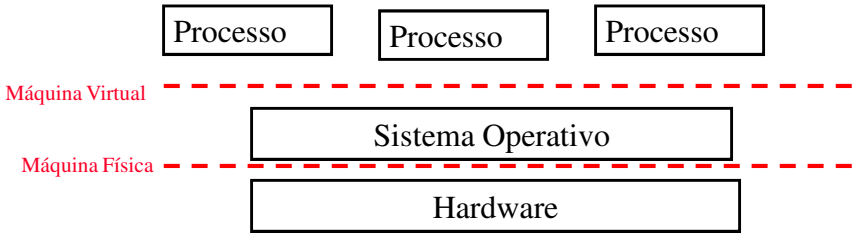
Exercício: Shell simples

```
while (TRUE){
    prompt();
    read_command (command, params);

    pid = fork ();
    if (pid == 0) {
        if (execv (command, params)==-1) {
           printf("Comando invalido\n.");
            exit(0);
        }
    } else if (pid > 0) {
        wait(&status);
    }
    else {
        printf ("Unable to fork"):
    }
}
```


2. Núcleo



Missão do Sistema Operativo

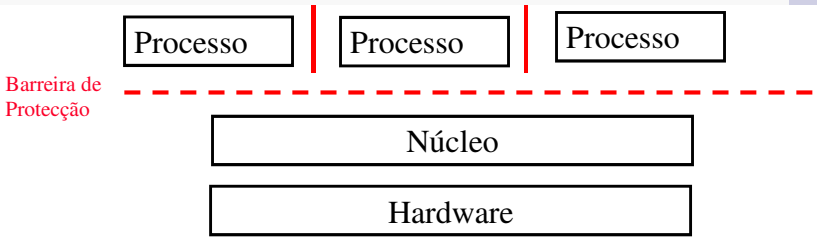


- Criar uma máquina virtual sobre a máquina física que ofereça os recursos lógicos básicos necessários ao desenvolvimento das aplicações
- Independente do hardware onde se executa

Como isolar Processos e Núcleo?

- Suporte do processador a diferentes modos de execução
  - Modo Núcleo e Modo Utilizador
- Em Modo Núcleo, o programa em execução pode:
  - Aceder a toda a memória
  - Executar qualquer instrução do processador
- Em Modo Utilizador:
  - Só é permitido aceder ao espaço de endereçamento atribuído ao processo em execução
  - Não é permitido executar instruções protegidas
    - Exemplos: Entrada/saída para periféricos ou inibir de interrupções

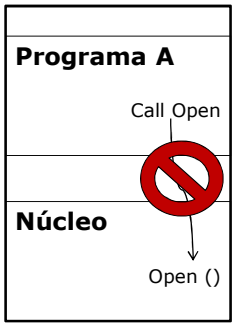
Como isolar Processos e Núcleo?



E então pode um Processo ter acesso fora do seu espaço de endereçamento?

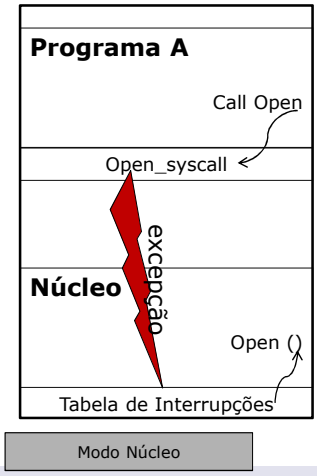
Chamando funções sistema (disponibilizadas pelo Núcleo)

Chamadas Sistema



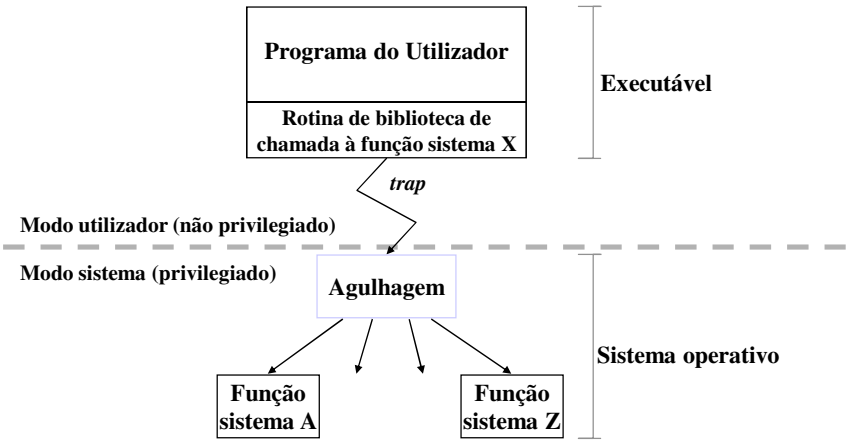
Memória do computador (podemos assumir endereçamento real)

Chamadas Sistema

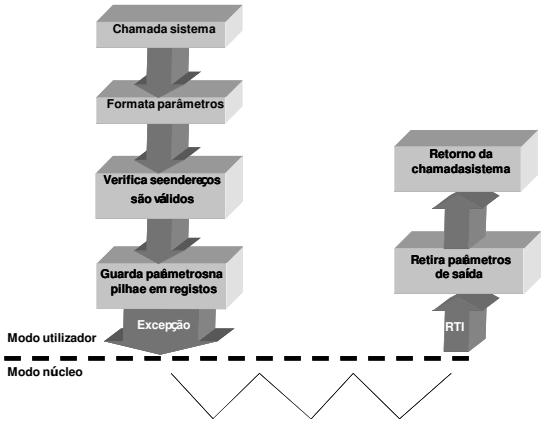


Memória do computador (podemos assumir endereçamento real)

Chamadas Sistema em Detalhe



Chamada Sistema em Detalhe



Protecção no Acesso aos Recursos

Protecção no Acesso aos Recursos em Unix

- Um processo tem associados dois identificadores que são atribuídos quando o utilizador efectua o login (se autentica) perante o sistema:
  - o número de utilizador UID - *user identification*
  - o número de grupo GID - *group identification*
- Os UID e GID são obtidos do ficheiro **`/etc/passwd`** no momento do *login*
- O UID e o GID são herdados pelos processos filhos
- *superuser* é um UID especial – zero
  - Normalmente está associado ao utilizador root (privilegiado).

Protecção no Acesso aos Recursos em Unix

- A protecção dos recursos em Unix é uma versão simplificada do modelo de Listas de Controlo de Acesso (ACL)
- Para um recurso (ficheiro, socket, etc.) a protecção é definida em três categorias:
  - Dono (*owner*): utilizador que normalmente criou o recurso
  - Grupo (*group*): conjunto de utilizadores com afinidades de trabalho que justificam direitos semelhantes
  - Restantes utilizadores (*world*)

SetUID

- Mecanismo de Set UID (SUID) – permite alterar dinamicamente o utilizador
- Duas variantes: bit de setuid, ou função sistema setuid

Bit SetUID

- No ficheiro executável pode existir uma indicação especial que na execução do exec provoca a alteração do uid
- O processo assume a identidade do dono do ficheiro durante a execução do programa.
- Exemplo: comando **`passwd`**
- Operação crítica para a segurança

Funções Sistema de identificação

- *Real UID e GID* – UID e GID originais do processo
- *Effective UID e GID* – usado para verificar permissões de acesso e que pode ter sido modificado pelo setuid

`getpid()` - devolve a identificação do processo

`getuid()`, `getgid()`

devolvem a identificação real do utilizador

`geteuid()`, `getegid()`

devolvem a identificação efectiva do utilizador

`setuid(uid)`, `setgid(gid)`

altera a identificação efectiva do utilizador para uid e gid

só pode ser invocada por processos com privilégio de superutilizador

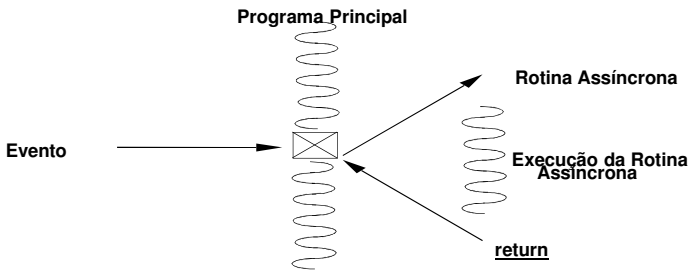
Eventos

Rotinas Assíncronas para Tratamento de acontecimentos assíncronos e excepções

Rotinas Assíncronas

- Certos acontecimentos devem ser tratados pelas aplicações, embora não seja possível prever a sua ocorrência
  - Ex: Ctrl-C
  - Ex: Acção desencadeada por um timeout
- Como tratá-los na programação sequencial?
- Poder-se-ia lançar uma tarefa por acontecimento. Desvantagem?
- Alternativa: Rotinas assíncronas associadas aos acontecimentos (**eventos**)

Modelo de Eventos



- Semelhante a outro conceito...

Rotinas Assíncronas

```
RotinaAssincrona (Evento,Procedimento)
```

Tem de existir uma tabela com os eventos que o sistema pode tratar

Identificação do procedimento a executar assincronamente quando se manifesta o evento.

Signals – Acontecimentos Assíncronos em Unix

Signal	Causa
SIGALRM	O relógio expirou
SIGFPE	Divisão por zero
SIGINT	O utilizador carregou na tecla para interromper (normalmente o CNTL-C
SIGQUIT	O utilizador quer terminar o processo e provoca
SIGKILL	Signal para terminar o processo. Não pode ser tratado
SIGPIPE	O processo escreveu para um pipe que não tem receptores
SIGSEGV	Acesso a uma posição de memória inválida
SIGTERM	O utilizador pretende terminar ordeiramente o proc
SIGUSR1	Definido pelo utilizador
SIGUSR1	Definido pelo utilizador

Excepção

Interacção com o terminal

Desencadeado por interrupção HW

Explicitamente desencadeado por outro processo

- Definidos em **`signal.h`**

Tratamento dos Signals

3 Possibilidades:

- Terminar o processo.
- Ignorar signal.
  - Alguns signals como o SIGKILL não podem ser ignorados. Porquê?
- Correr rotina de tratamento (handler)
  - Associamos rotina de tratamento a signal pela função sistema `signal`

Cada signal tem um tratamento por omissão, que pode ser terminar ou ignorar

Chamada Sistema "Signal"

```
void (*signal (int sig, void (*func)(int))) (int);
```

A função retorna um ponteiro para função anteriormente associada ao signal

Identificador do signal para o qual se pretende definir um handler

Ponteiro para a função ou macro especificando:
- **SIG_DFL – acção por omissão**
- **SIG_IGN – ignorar o signal**

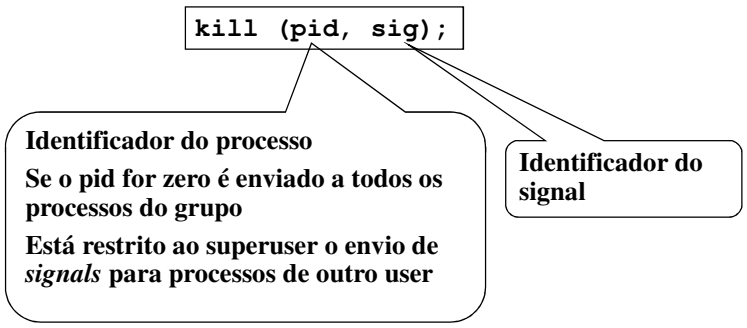
Parâmetro para a função de tratamento

Exemplo do tratamento de um Signal

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

apanhaCTRLC () {
   char ch;
   printf ("Quer de facto terminar a execucao?\n");
   ch = getchar();
   if (ch == 's') exit(0);
   else {
      printf ("Entao vamos continuar\n");
      signal (SIGINT, apanhaCTRLC);
   }
}

main () {
   signal (SIGINT, apanhaCTRLC);
   printf("Associou uma rotina ao signal SIGINT\n");
   for (;;)
     sleep (10);
}
```


Chamada Sistema Kill

- Envia um signal ao processo
- Nome enganador. Porquê?

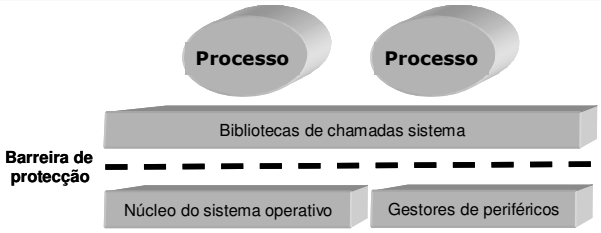


Outras funções associadas aos signals

- **`unsigned alarm (unsigned int segundos);`**
  - o *signal* SIGALRM é enviado para o processo depois de decorrerem o número de segundos especificados. Se o argumento for zero, o envio é cancelado.
- **`pause();`**
  - aguarda a chegada de um *signal*
- **`unsigned sleep (unsigned int segundos);`**
  - A função faz um alarm e bloqueia-se à espera do signal

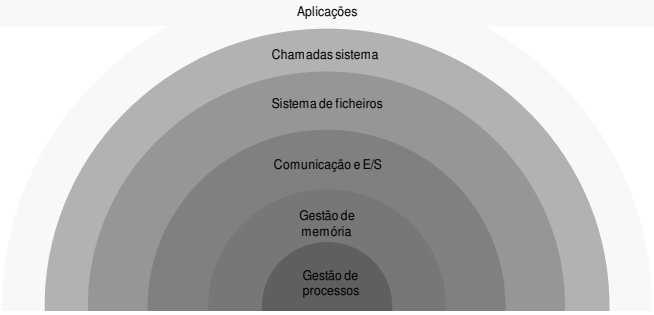
Como está organizado o Núcleo?

Estrutura Monolítica



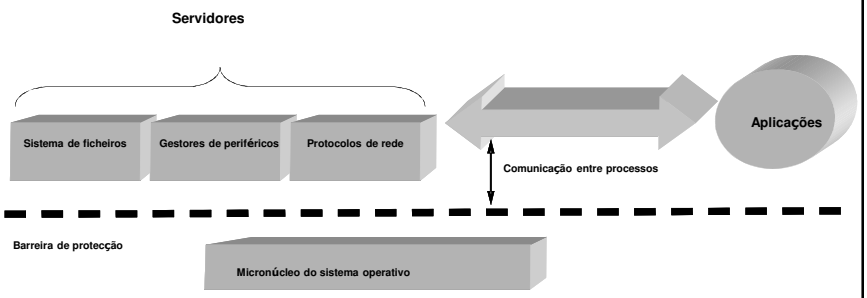
- Um único sistema
- Internamente organizado em módulos
- Estruturas de dados globais
- Problema: como dar suporte à evolução?
  - Em particular, novos periféricos
- Solução para este caso particular: gestores de dispositivos (*device drivers*)
- Problemas?

Sistemas em Camadas



- Cada camada usa os serviços da camada precedente
- Fácil modificar código de uma camada
- Mecanismos de protecção → maior segurança e robustez
- Influenciou arquitecturas como Intel
- Desvantagem principal?

Micro-Núcleo

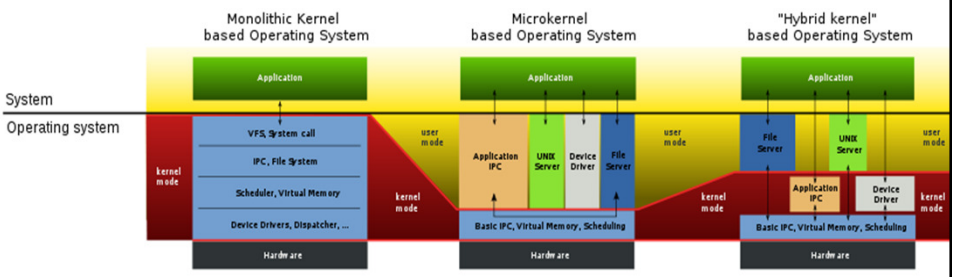


Micro-Núcleo

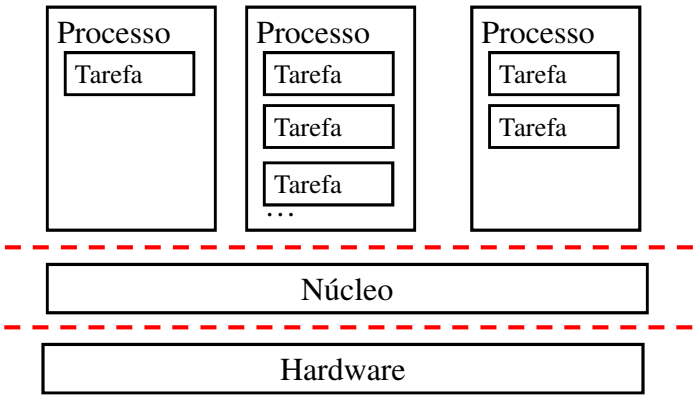
Separação entre:

- Um micro-núcleo de reduzidas dimensões e que só continha o essencial do sistema operativo:
  - Gestão de fluxos de execução - threads
  - Gestão dos espaços de endereçamento
  - Comunicação entre processos
  - Gestão das interrupções
- Servidores sistema que executavam em processos independentes a restante funcionalidade:
  - Gestão de processos
  - Memória virtual
  - Device drivers
  - Sistema de ficheiros

Micro-Núcleo vs Monolítico

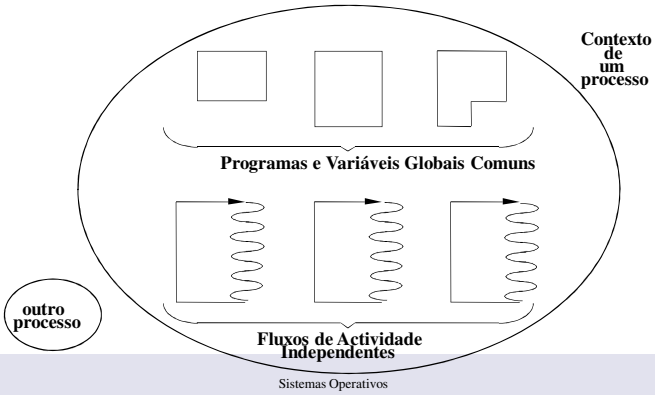


3. Tarefas (Threads)



Tarefas

- Mecanismo simples para criar fluxos de execução independentes, partilhando um contexto comum



Tarefas vs. Processos

- Porque não usar processos?
  - Processos obrigam ao isolamento (espaços de endereçamentos disjuntos) → dificuldade em partilhar dados (mas não impossível… exemplos?)
  - Eficiência na criação e comutação

Tarefas: Exemplos de Utilização

- Servidor (e.g., web)
- Aplicação cliente de correio electrónico
- Quais as tarefas em cada caso?

Modelos Multitarefa no Modelo Computacional

- Operações sobre as Tarefas

```
IdTarefa = CriarTarefa(procedimento);
```

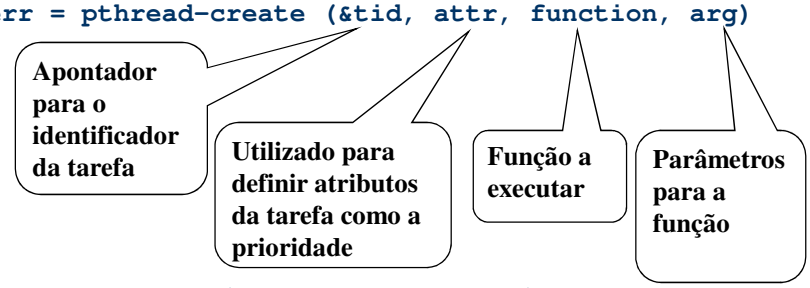
A tarefa começa a executar o procedimento dado como parâmetro e que faz parte do programa previamente carregado em memória

```
EliminarTarefa (IdTarefa);
EsperaTarefa (IdTarefa)
```

Bloqueia a tarefa à espera da terminação de outra tarefa ou da tarefa referenciada no parâmetro Idtarefa

Interface POSIX

```
err = pthread-create (&tid, attr, function, arg)
```




```
pthread_exit(void *value_ptr)
```

```
int pthread_join(pthread_t thread, void **value_ptr)
```

- se a tarefa alvo não terminou a tarefa continua, senão bloqueia-se

Exemplo (sequencial)

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>
#define N 5
#define TAMANHO 10

int buffer [N] [TAMANHO];
int nsomas;

void *soma_linha (int *linha) {
  int c, soma=0;
  int *b = linha;
  for (c = 0; c < TAMANHO - 1; c++) {
    soma += b[c];
    nsomas++;
  }

  b[c]=soma; /* soma->ult.col.*/
  return NULL;
}
```

```
int main (void) {
  int i,j;

  for (i=0; i<N; i++){
    for (j=0; j< TAMANHO - 1; j++)
      buffer[i] [j] =rand()%10;
  }

 for (i=0; i< N; i++)
   soma_linha(buffer[i]);


 imprimeResultados(buffer);

exit(0);
}
```

Exemplo (paralelo)

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>
#define N 5
#define TAMANHO 10

int buffer [N] [TAMANHO];
int nsomas;

void *soma_linha (int *linha) {
  int c, soma=0;
  int *b = linha;
  for (c = 0; c < TAMANHO - 1; c++) {
    soma += b[c];
    nsomas++;
  }

  b[c]=soma; /* soma->ult.col.*/
  return NULL;
}
```


Exemplo (paralelo)

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>
#define N 5
#define TAMANHO 10

int buffer [N] [TAMANHO];
int nsomas;

void *soma_linha (int *linha) {
  int c, soma=0;
  int *b = linha;
  for (c = 0; c < TAMANHO - 1; c++) {
    soma += b[c];
    nsomas++;
  }

  b[c]=soma; /* soma->ult.col.*/
  return NULL;
}
```

```
int main (void) {
  int i,j;
  pthread_t tid[N];

  /* inicializa buffer … */

 for (i=0; i< N; i++){
   if(pthread_create (&tid[i], 0,soma_linha,
                      (void *) buffer[i])== 0) {
      printf ("Criada a tarefa %d\n", tid[i]);
   }
   else {
     printf("Erro na criação da tarefa\n");
     exit(1);
   }
 }

 for (i=0; i<N; i++){
   pthread_join (tid[i], NULL);
 }
 printf ("Terminaram todas as threads\n");

 imprimeResultados(buffer);

 exit(0);
}
```

Programação num ambiente multitarefa

- As tarefas partilham o mesmo espaço de endereçamento e portanto têm acesso às mesmas variáveis globais.
- A modificação e teste das variáveis globais tem de ser efectuada com precauções especiais para evitar erros de sincronização.
- Veremos no cap. 4 a forma de resolver estes problema com objectos de sincronização.

Alternativas de Implementação

- Tarefas-núcleo
- Tarefas-utilizador (pseudotarefas)
  - Projecto da cadeira

Pseudotarefas (Tarefas-Utilizador)

- As tarefas implementadas numa biblioteca de funções no espaço de endereçamento do utilizador.
- Ideia proveniente das linguagens de programação.
- Núcleo apenas "vê" um processo.
- Processo guarda lista de tarefas, respectivo contexto

Departamento de Engenharia Informática

Pseudotarefas (Tarefas-Utilizador)

- A comutação entre tarefas explícita → função **`thread_yield`**
  - Pode ser contornado usando interrupções ("preempção")
- Problema: e se uma tarefa faz chamada bloqueante?
- Solução?

Tarefas-Núcleo (ou Tarefas Reais)

- Implementadas no núcleo do SO
  - Mais comuns
- Lista de tarefas e respectivo contexto são mantidos pelo núcleo

Comparação Tarefas Utilizador e Núcleo

1. Capacidade de utilização em diferentes SOs?
2. Velocidade de criação e comutação? (vs. processos?)
3. Tirar partido de execução paralela em multiprocessadores?
4. Aproveitamento do CPU quando uma tarefa bloqueia (ex: ler do disco)?