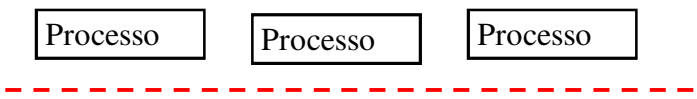
Processos, tarefas e núcleo

Sistemas Operativos

2012 / 2013

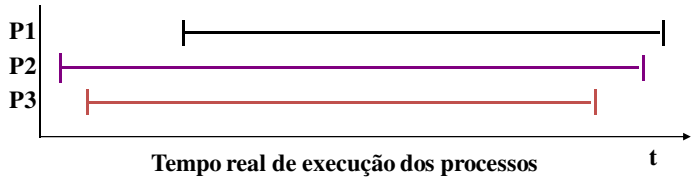
1. Processos



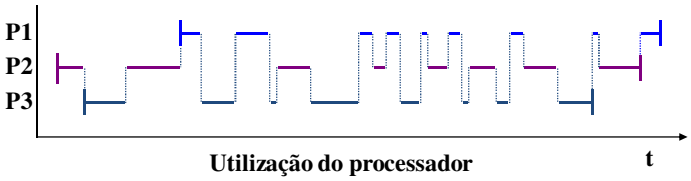
Multiprogramação

- 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



Tempo real de execução dos processos

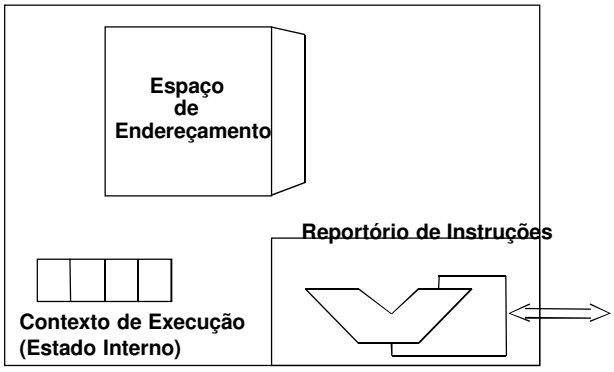


Utilização do processador

Processos vs. Programas

- Programa = Fich. executável (sem actividade)
- Um processo é um objecto do sistema operativo que suporta a execução dos programas
- Um processo pode, durante a sua vida, executar diversos programas
- Um programa ou partes de um programa podem ser partilhados por diversos processos
  - ex.: biblioteca partilhadas, as DLL no Windows

Processo Como Uma Máquina Virtual

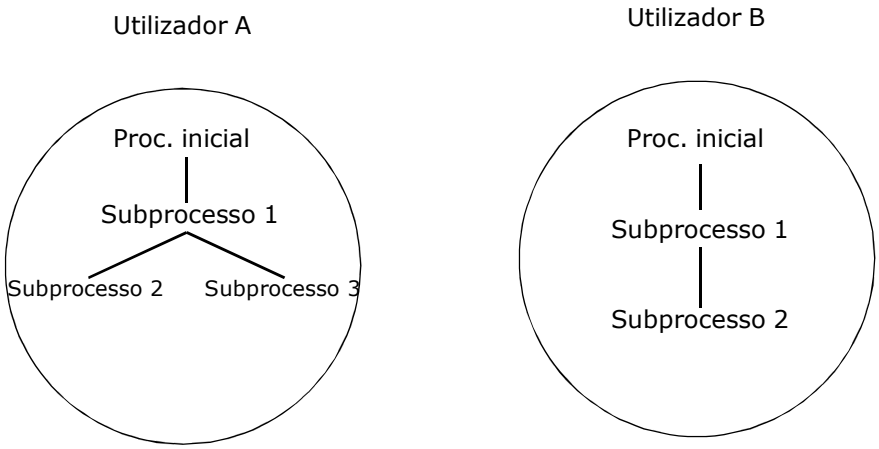


Elementos principais da máquina virtual que o SO disponibiliza aos processos

Processo Como Uma Máquina Virtual

- Tal como um processador um processo tem:
  - Espaço de endereçamento (virtual):
    - Conjunto de posições de memória acessíveis
    - Código, dados, e pilha
    - Dimensão variável
  - Reportório de instruções:
    - As instruções do processador executáveis em modo utilizador
    - As funções do sistema operativo
  - Contexto de execução (estado interno):
    - Valor dos registos do processador
    - Toda a informação necessária para retomar a execução do processo
    - Memorizado quando o processo é retirado de execução

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

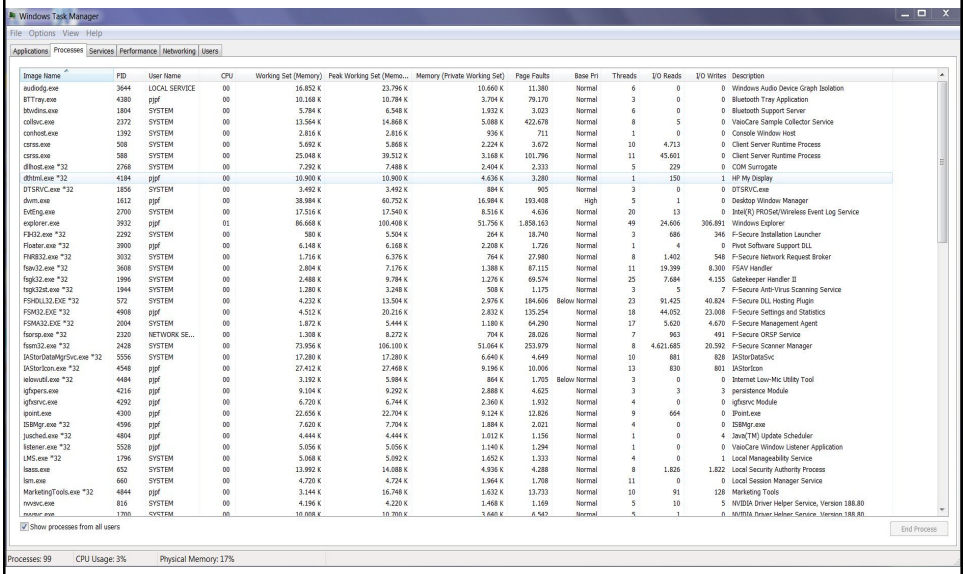
- Operações – Funções sistema que actuam sobre os processos
  - Criar
  - Eliminar
  - Esperar pela terminação de subprocesso

Exemplo: Unix

```
ps -el | 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
```

```
    ps     displays information about a selection of the
    active processes.
    e      select all processes
    l      long format
```

Exemplo: Windows



Criação de um processo

IdProcesso = CriarProc (Código, Prioridade,...)

Quando a criação tem sucesso o sistema atribui um identificador interno (PID) ao processo que é retornado na função

A função tem frequentemente diversos parâmetros: a prioridade, canais de entrada/saída, ...

Na criação de um processo tem de ficar definido qual é o programa que o processo vai executar. Normalmente é especificado um ficheiro contendo um programa executável.

Eliminação de processos

- Eliminação do processo quando o seu programa termina, libertando todos os recursos e estruturas de dados detidas pelo processo
  - Sair ([Estado])
- Eliminação de outro processo
  - EliminarProc (IdProcesso)

O processo cujo identificador é passado como parâmetro é eliminado. O núcleo do SO valida se o processo que invoca esta função tem privilégios para a poder executar

Terminação do Processo Filho

- Em numerosas situações o processo pai pode querer bloquear-se esperando a terminação de um processo filho
- Estado = EsperarTerminacao (Idprocesso)

O processo pai pode esperar por um processo específico ou genericamente por qualquer processo

Modelo de Segurança

- Um processo em execução tem de estar associado a um Utilizador
  - Entidade que pode ser responsabilizada pelos seus actos
- Os utilizadores são representados no sistema por um código que os identifica
  - User IDentifier – UID
- Para facilitar a partilha o utilizador pode pertencer a um grupo ou grupos de utilizadores
  - Identificado por um GID

Controlo dos Direitos de Acesso

- Autorização - operação que valida os direitos do utilizador sobre um recurso antes deste poder executar uma operação sobre ele.
- A autorização baseia-se conceptualmente numa Matriz de Direitos de Acesso

		Objectos	
Utilizadores	**1**	**2**	**3**
1	Ler	-	Escrever
2	-	Ler/ Escrever	-
3	-	-	Ler

- Para um dado objecto a coluna da matriz define a Lista de Direitos de Acesso (ACL)
- Para um dado utilizador a linha respectiva define todos os seus direitos normalmente designados por Capacidade

UNIX – PROCESSOS

(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) e 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 designado por processo pai.
  - Quando o processo pai termina os subprocessos continuam a executar-se, são adoptados pelo processo de inicialização (pid = 1)
- Os processos têm prioridades variáveis.
  - Veremos as regras de escalonamento mais adiante.

Processos em Unix

- Espaço de endereçamento em modo Utilizador
  - Organiza-se em três zonas que no Unix original se designavam por segmentos:
    - texto - código do programa
    - dados - espaço de dados do programa
    - pilha (stack)
- Espaço de endereçamento em modo Núcleo
  - No interior do núcleo existe uma zona de dados para cada processo que contem o seu contexto
  - Uma pilha para execução do processo em modo núcleo.

Processos em Unix

- Cada processo também tem associado um contexto de execução acessível em modo utilizador e que contém diversas variáveis úteis para os programas utilitários ou para as aplicações.
- Exemplo:
  - HOME=/usr/pjpf
  - SHELL=/bin/csh
  - USER=pjpf
  - PATH=/usr/pjpf/bin/:/usr/local/bin:/bin
- Este contexto é herdado do processo pai e pode ser modificado livremente porque reside no espaço utilizador.
- Nos programas em C é acessível através do parâmetro do main ou de uma variável externa:
  - main (arc, arv, envp)
  - extern char **environ

Criação de um Processo

id = fork()

A função não tem parâmetros, em particular o ficheiro a executar. A imagem do novo processo é uma cópia da do criador.

O contexto do processo pai é copiado para o filho

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);
    }
}
```


Execução de um Programa

- O fork apenas permite lançar processo com o mesmo código → problemas?
- A função exec permite substituir a imagem do processo onde é invocada pela contida num ficheiro executável.
- Não há retorno numa chamada com sucesso.
- Parâmetros: valores que são passados para os parâmetros de entrada na função main do código a executar.
- Os ficheiros mantêm-se abertos.

Execução de um Programa

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

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

Caminho de acesso ao ficheiro executável

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

Exemplo de Exec

```
main ()
{
    int pid;

    pid = fork ();
    if (pid == 0) {
        execl ("/bin/who", "who", 0);
        /* controlo deveria ser transferido para o novo
             programa  */
        printf ("Erro no execl\n");
        exit (-1);
    } else {
        /* algoritmo do proc. pai */
    }
}
```

Por convenção o arg0 é o nome do programa

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);









}
```

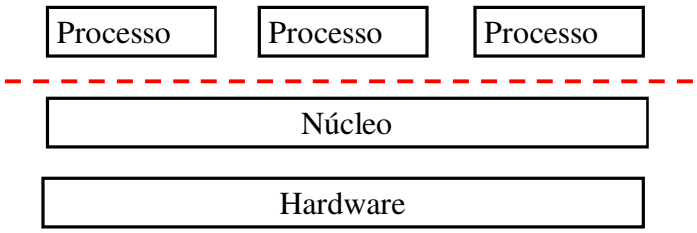
Exercício: Shell simples

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

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

    } else if (pid > 0) {

    }
    else {

    }
}
```

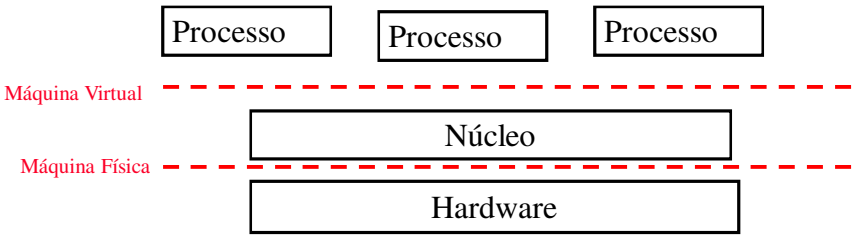

Exercício: Shell simples

```
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

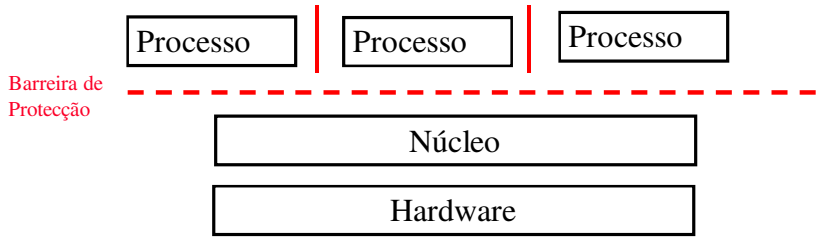


- Criar uma máquina virtual sobre a máquina física que ofereça os recursos lógicos básicos necessários ao desenvolvimento das aplicações
- Independente do hardware onde se executa

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

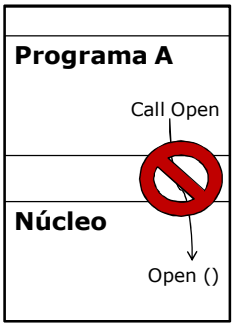
Como isolar Processos e Núcleo?



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

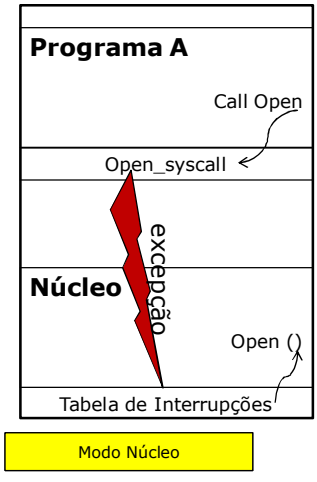
Chamando funções sistema (disponibilizadas pelo Núcleo)

Chamadas Sistema



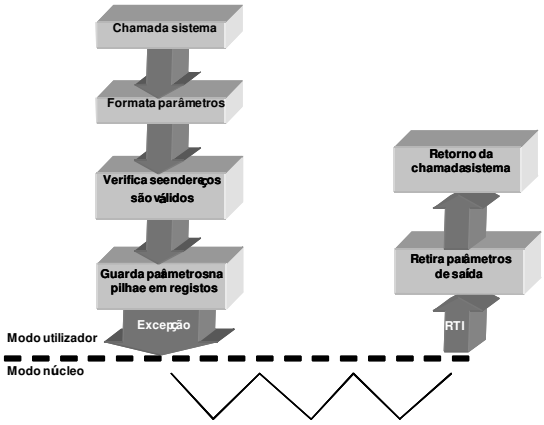
Memória do computador (podemos assumir endereçamento real)

Chamadas Sistema



Memória do computador (podemos assumir endereçamento real)

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*)

Sistemas Operativos – DEI - IST

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, herdados do processo pai
- *Effective UID e GID*
  - usados 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

...olo

Assumam que processo pai é lançado pela shell do UID=1234

getuid()=?
geteuid()=?

```
int main() {
  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 ("Una
    }
  }
}
```

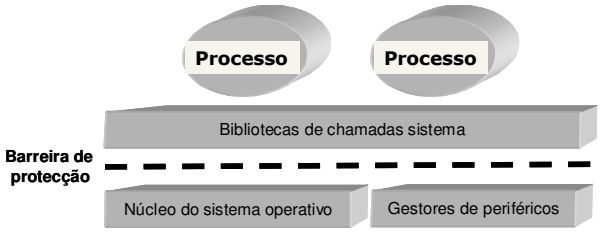
getuid()=?
geteuid()=?

Assumam que ficheiro indicado em *command* é possuido pelo Utilizador com UID=9876

Dentro do programa executado,
getuid()=?
geteuid()=?

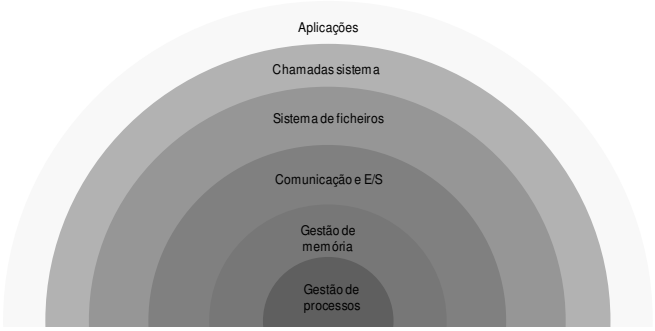
Como está organizado o Núcleo?

Estrutura Monolítica



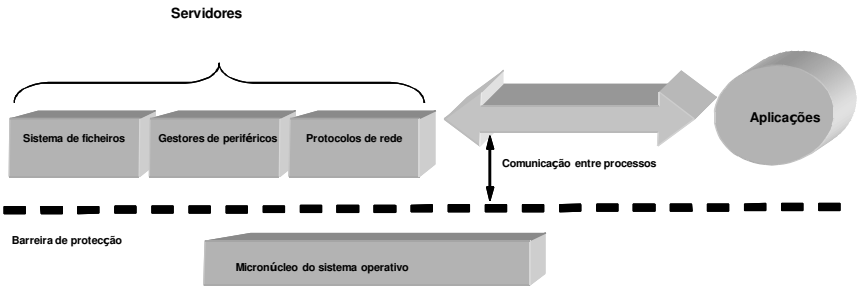
- Um único sistema
- Internamente organizado em módulos
- Estruturas de dados globais
- Problema: como dar suporte à evolução?
  - Em particular, novos periféricos
- Solução para este caso particular: gestores de dispositivos (*device drivers*)
- Problemas?

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

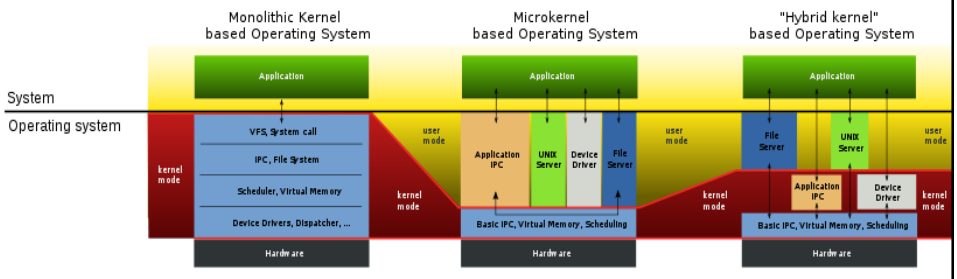


Micro-Núcleo

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



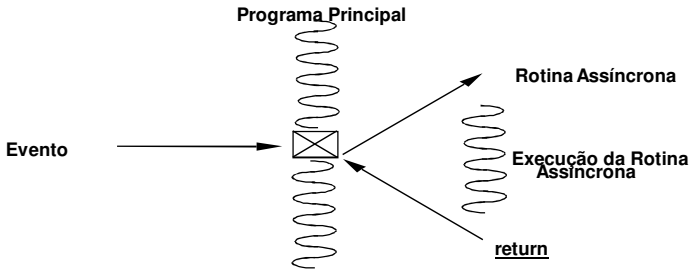
Eventos

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 CTRL-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
SIGUSR2	Definido pelo utilizador

Excepção

Interacção com o terminal

Desencadeado por interrupção HW

Explicitamente desencadeado por outro processo

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

Tratamento dos Signals

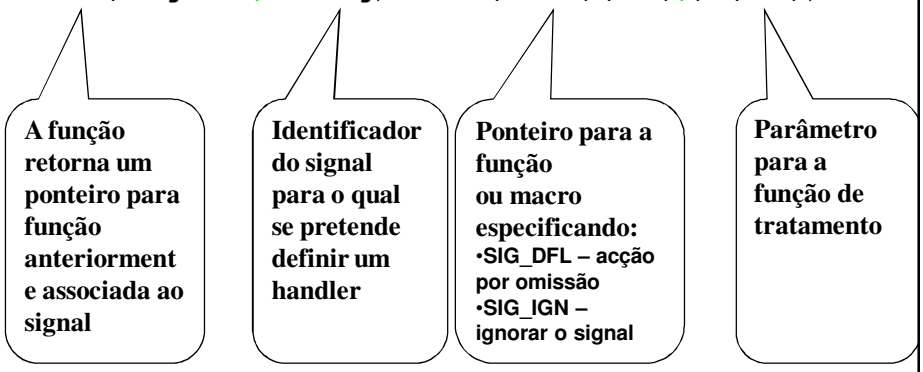
3 Possibilidades:

- 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);
```

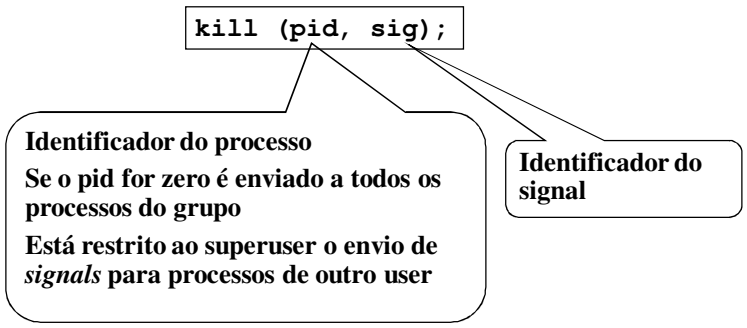


Exemplo do tratamento de um Signal

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

void apanhaCTRLC (int x) {
   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

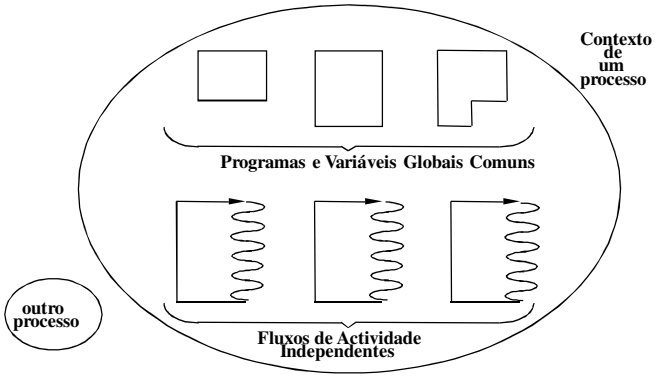
- **`unsigned alarm (unsigned int segundos);`**
  - o *signal* SIGALRM é enviado para o processo depois de decorrerem o número de segundos especificados. Se o argumento for zero, o envio é cancelado.
- **`pause();`**
  - aguarda a chegada de um *signal*
- **`unsigned sleep (unsigned int segundos);`**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

TAREFAS (THREADS)

Múltiplos fluxos de execução no mesmo processo

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);

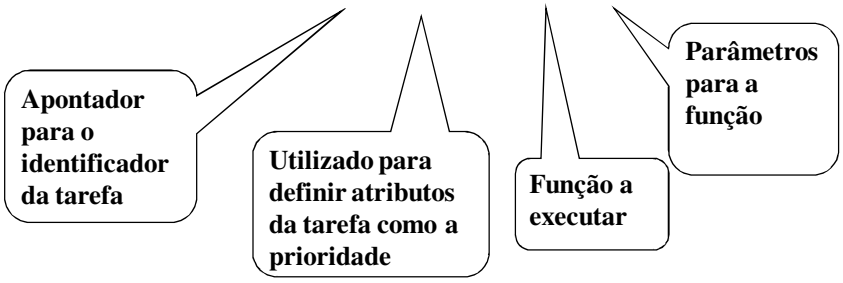
A tarefa começa a executar o procedimento dado como parâmetro e que faz parte do programa previamente carregado em memória

  - 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)
  - suspende a tarefa invocadora até “pthread_t thread“ terminar; continua a execução caso “pthread_t thread“ já tenha terminado

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)

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
- Problema: e se uma tarefa faz chamada bloqueante?
  - Solução?

Pseudotarefas (Tarefas-Utilizador): Comutação entre tarefas

- Pode ser explícita
  - Tarefa que quer ceder processador a outra tarefa chama função **`thread-yield`**
- Pode ser implícita
  - Ocorrência de signal periódico (SIGALM) é tratada por rotina que chama **`thread-yield`**

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

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

PROCESSOS NO WINDOWS 2000

Processos – Windows

- Um processo é um contentor de recursos usados pelas tarefas
- Os fluxos de execução são as threads
- Processo → uma ou mais threads

Processos

- Um processo em Windows 2000 é constituído por:
  - Um espaço de endereçamento
  - Um programa executável
  - Pelo menos uma tarefa
  - Uma lista de referências (handles) para vários **objectos** (quaisquer recursos do sistema)
  - Um contexto de segurança
  - Um identificador único do processo - process ID

Threads

- Tarefas reais.
- Componentes fundamentais:
  - Os registos do CPU que representam o estado do processador
  - Duas pilhas (*stacks*), uma para execução em modo núcleo e outra para execução em modo utilizador
  - Uma zona de memória privada (*thread-local storage* - TLS) para uso pelos subsistemas e DLLs
  - Um identificador único - thread ID

Fibers

- Pseudotarefas geridas no espaço de endereçamento do utilizador.
- Uma thread pode ter múltiplas fibers.
- Fibers não são vistas pelo núcleo
- As fibers são criadas e comutadas explicitamente com chamadas à biblioteca Win32 mas que não produzem chamadas ao sistema.

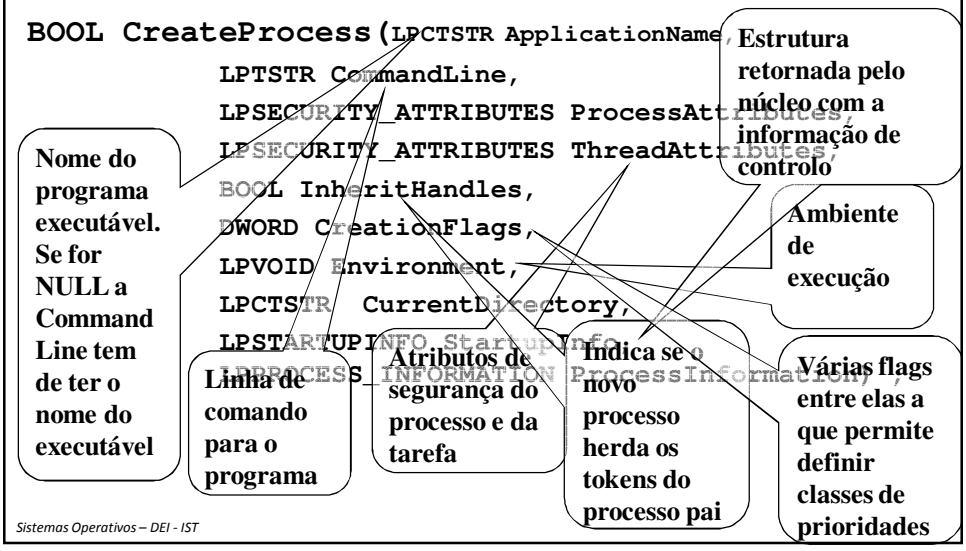
Jobs

- Grupo de processos
  - Permite gestão uniforme (e.g., terminar em conjunto)
- Um processo só pode ser associado a um job e em principio todos os seus descendentes pertencem ao mesmo job

Segurança

- O contexto de segurança de um processo ou de uma thread é um objecto designado *Access Token*
- Um *Access Token* regista os utilizadores, grupos, máquinas, e domínios que estão associados ao processo.
- Sempre que é acedido um objecto no sistema o *executive* valida o token contra uma ACL
- Acesso concedido se não exisitir nenhuma recusa, e existir pelo menos uma permissão num dos utilizadores, grupos, etc.

Criação de processos Win32



Criação de processos Win32

- Diferenças vs. fork+exec:
  - No Windows não se cria automaticamente uma relação pai-filho. Embora o processo pai fique com um handle para o filho.
  - Um processo tem associado uma thread (main thread).
  - Na criação do processo pode definir-se a classe de prioridade a que as threads do processo ficam associadas.
  - A criação com sucesso retorna um valor diferente de zero.

Criação de processos Win32

```
#include <windows.h>
#include <stdio.h>
#include <string.h>
STARTUPINFO startInfo;
PROCESS_INFORMATION processInfo;
…
strcpy(lpCommandLine,
  "C:\\WINNT\\SYSTEM32\\NOTEPAD.EXE temp.txt");
ZeroMemory(&startInfo, sizeof(startInfo));
startInfo.cb = sizeof(startInfo);
If(!CreateProcess(NULL, lpCommandLine, NULL, NULL, FALSE,
  HIGH_PRIORITY_CLASS  CREATE_NEW_CONSOLE,
  NULL, NULL, &startInfo, &processInfo))  {
       fprintf(stderr, "CreateProcess failed on error %d\n",
       GetLastError());
       ExitProcess(1);
};
```


Eliminação de Processos

- Existem três formas para terminar um processo
  - Chamada à função ExitProcess que autotermina o processo
  - Chamada à função TerminateProcess que permite a um processo com o requerido privilégio terminar outro processo
  - Terminando todas as threads de um processo

Eliminação de Processos

```
void ExitProcess (UINT uExitCode);
```

- Informa todas as DLLs que o processo termina.
- Fecha todos os handles do processo
- Termina todas as threads

Código de terminação

```
bool TerminateProcess (Handle Processo, UINT uExitCode);
```


Criação Thread

```
HANDLE CreateThread (
        LPSECURITY_ATTRIBUTES ThreadAttr,
        DWORD stakSz,
        LPTHREAD_START_ROUTINE StrtAddr,
        LPVOID Parm,
        DWORD CreatFlgs,
        LPDWORD ThreadId);
```

Handle para a thread ou NULL se falhou

Atributos de segurança

Tamanho do stack

Endereço da função inicial

Um parâmetro que pode ser passado à thread

Esperar Pela Terminação de Subprocesso

- **`WaitForSingleObject(handle, timeout)`**
- Função genérica de espera sobre um objecto (entidade do sistema operativo)