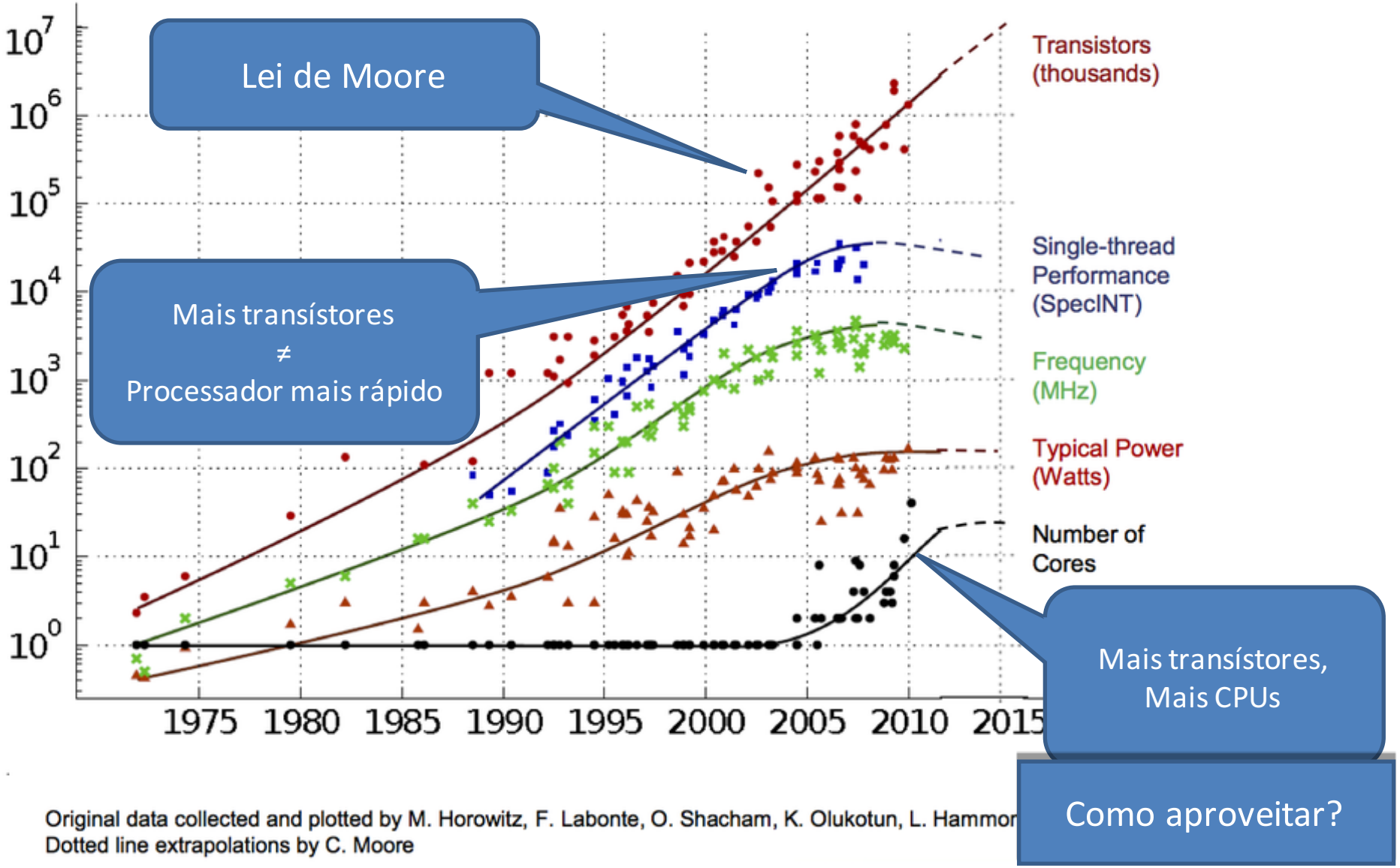
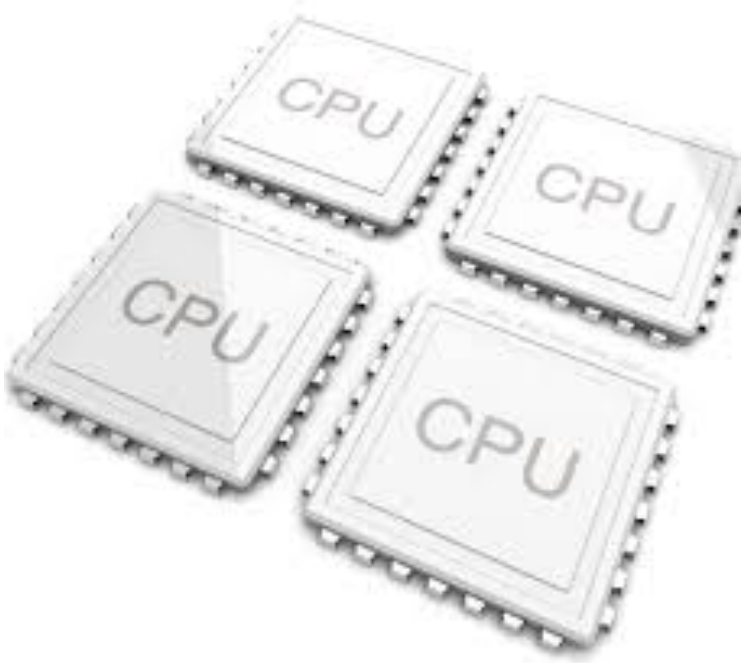
O fim dos “almoços gratis”



Programação paralela

- Permite explorar processadores múltiplos
  - Incluindo os dual-cores, quad-cores, etc.



Outras razões para optar por programação paralela?

- Interação com periféricos lentos
  - Enquanto periférico demora a responder a um fluxo de execução, outro fluxo paralelo pode continuar a fazer progresso
- Idem para programas interativos
  - Enquando um fluxo de execução espera por ação do utilizador, outros podem progredir em fundo

Ou seja, programar com múltiplas tarefas faz sentido mesmo em máquinas *single-cpu!*

Próximas aulas:
Dois níveis de programação paralela (processos e tarefas)

Introdução à programação com processos

Sistemas Operativos

2018 / 2019

Multiprogramação

- Execução, em paralelo, de múltiplos programas na mesma máquina

- Cada instância de um programa em execução denomina-se um **processo**

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

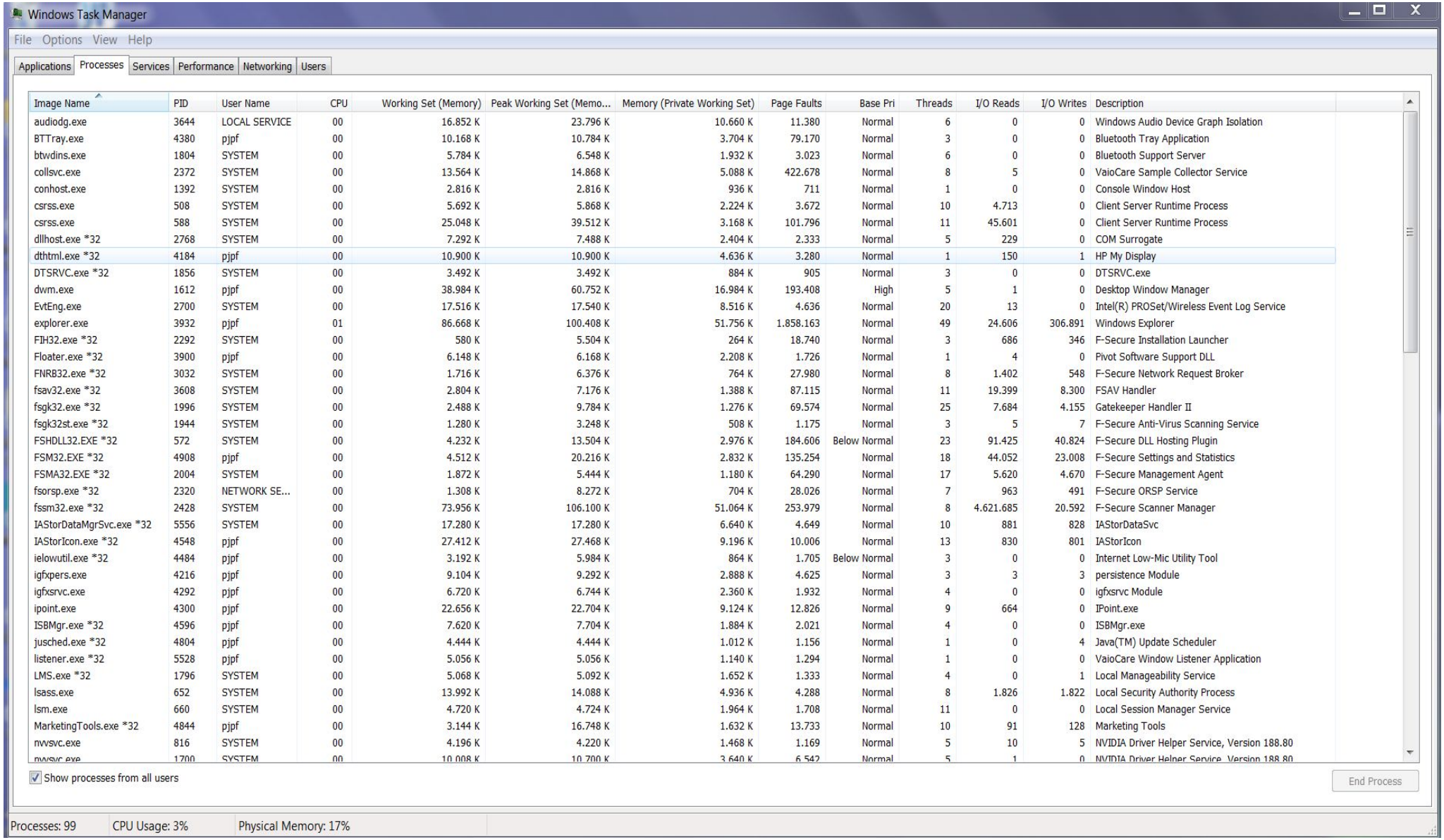
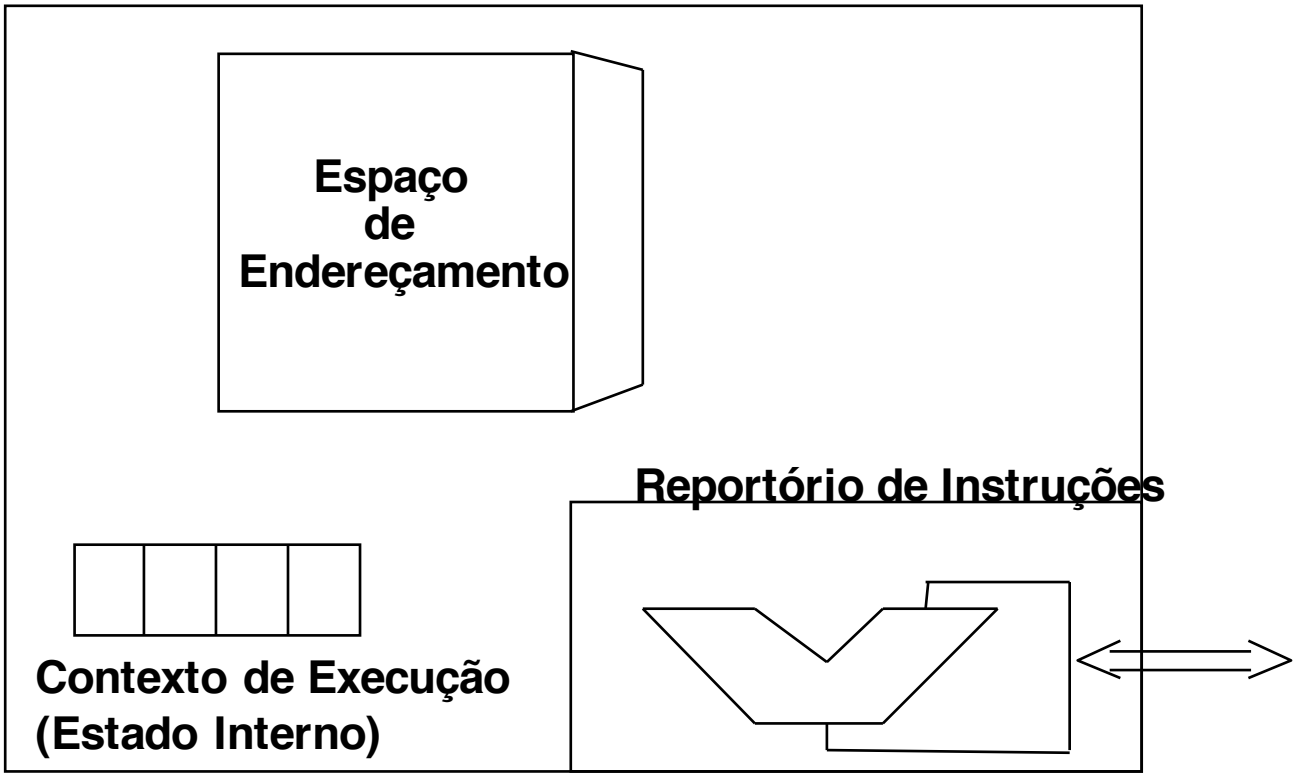


Image Name	PID	User Name	CPU	Working Set (Memory)	Peak Working Set (Memo...	Memory (Private Working Set)	Page Faults	Base Pri	Threads	I/O Reads	I/O Writes	Description
audiodg.exe	3644	LOCAL SERVICE	00	16.852 K	23.796 K	10.660 K	11.380	Normal	6	0	0	Windows Audio Device Graph Isolation
BTTray.exe	4380	pjpf	00	10.168 K	10.784 K	3.704 K	79.170	Normal	3	0	0	Bluetooth Tray Application
btwdins.exe	1804	SYSTEM	00	5.784 K	6.548 K	1.932 K	3.023	Normal	6	0	0	Bluetooth Support Server
collsvc.exe	2372	SYSTEM	00	13.564 K	14.868 K	5.088 K	422.678	Normal	8	5	0	VaioCare Sample Collector Service
conhost.exe	1392	SYSTEM	00	2.816 K	2.816 K	936 K	711	Normal	1	0	0	Console Window Host
csrss.exe	508	SYSTEM	00	5.692 K	5.868 K	2.224 K	3.672	Normal	10	4.713	0	Client Server Runtime Process
csrss.exe	588	SYSTEM	00	25.048 K	39.512 K	3.168 K	101.796	Normal	11	45.601	0	Client Server Runtime Process
dllhost.exe *32	2768	SYSTEM	00	7.292 K	7.488 K	2.404 K	2.333	Normal	5	229	0	COM Surrogate
dthtml.exe *32	4184	pjpf	00	10.900 K	10.900 K	4.636 K	3.280	Normal	1	150	1	HP My Display
DTSRVC.exe *32	1856	SYSTEM	00	3.492 K	3.492 K	884 K	905	Normal	3	0	0	DTSRVC.exe
dwm.exe	1612	pjpf	00	38.984 K	60.752 K	16.984 K	193.408	High	5	1	0	Desktop Window Manager
EvtEng.exe	2700	SYSTEM	00	17.516 K	17.540 K	8.516 K	4.636	Normal	20	13	0	Intel(R) PROSet/Wireless Event Log Service
explorer.exe	3932	pjpf	01	86.668 K	100.408 K	51.756 K	1.858.163	Normal	49	24.606	306.891	Windows Explorer
FIH32.exe *32	2292	SYSTEM	00	580 K	5.504 K	264 K	18.740	Normal	3	686	346	F-Secure Installation Launcher
Floater.exe *32	3900	pjpf	00	6.148 K	6.168 K	2.208 K	1.726	Normal	1	4	0	Pivot Software Support DLL
FNRB32.exe *32	3032	SYSTEM	00	1.716 K	6.376 K	764 K	27.980	Normal	8	1.402	548	F-Secure Network Request Broker
fsav32.exe *32	3608	SYSTEM	00	2.804 K	7.176 K	1.388 K	87.115	Normal	11	19.399	8.300	FSAV Handler
fsgk32.exe *32	1996	SYSTEM	00	2.488 K	9.784 K	1.276 K	69.574	Normal	25	7.684	4.155	Gatekeeper Handler II
fsgk32st.exe *32	1944	SYSTEM	00	1.280 K	3.248 K	508 K	1.175	Normal	3	5	7	F-Secure Anti-Virus Scanning Service
FSHDLL32.EXE *32	572	SYSTEM	00	4.232 K	13.504 K	2.976 K	184.606	Below Normal	23	91.425	40.824	F-Secure DLL Hosting Plugin
FSM32.EXE *32	4908	pjpf	00	4.512 K	20.216 K	2.832 K	135.254	Normal	18	44.052	23.008	F-Secure Settings and Statistics
FSMA32.EXE *32	2004	SYSTEM	00	1.872 K	5.444 K	1.180 K	64.290	Normal	17	5.620	4.670	F-Secure Management Agent
fsorsp.exe *32	2320	NETWORK SE...	00	1.308 K	8.272 K	704 K	28.026	Normal	7	963	491	F-Secure ORSP Service
fssm32.exe *32	2428	SYSTEM	00	73.956 K	106.100 K	51.064 K	253.979	Normal	8	4.621.685	20.592	F-Secure Scanner Manager
IAStorDataMgrSvc.exe *32	5556	SYSTEM	00	17.280 K	17.280 K	6.640 K	4.649	Normal	10	881	828	IAStorDataSvc
IAStorIcon.exe *32	4548	pjpf	00	27.412 K	27.468 K	9.196 K	10.006	Normal	13	830	801	IAStorIcon
ielowutil.exe *32	4484	pjpf	00	3.192 K	5.984 K	864 K	1.705	Below Normal	3	0	0	Internet Low-Mic Utility Tool
igfxpers.exe	4216	pjpf	00	9.104 K	9.292 K	2.888 K	4.625	Normal	3	3	3	persistence Module
igfxsrvc.exe	4292	pjpf	00	6.720 K	6.744 K	2.360 K	1.932	Normal	4	0	0	igfxsrvc Module
ipoint.exe	4300	pjpf	00	22.656 K	22.704 K	9.124 K	12.826	Normal	9	664	0	IPoint.exe
ISBMgr.exe *32	4596	pjpf	00	7.620 K	7.704 K	1.884 K	2.021	Normal	4	0	0	ISBMgr.exe
jusched.exe *32	4804	pjpf	00	4.444 K	4.444 K	1.012 K	1.156	Normal	1	0	4	Java(TM) Update Scheduler
listener.exe *32	5528	pjpf	00	5.056 K	5.056 K	1.140 K	1.294	Normal	1	0	0	VaioCare Window Listener Application
LMS.exe *32	1796	SYSTEM	00	5.068 K	5.092 K	1.652 K	1.333	Normal	4	0	1	Local Manageability Service
lsass.exe	652	SYSTEM	00	13.992 K	14.088 K	4.936 K	4.288	Normal	8	1.826	1.822	Local Security Authority Process
lsm.exe	660	SYSTEM	00	4.720 K	4.724 K	1.964 K	1.708	Normal	11	0	0	Local Session Manager Service
MarketingTools.exe *32	4844	pjpf	00	3.144 K	16.748 K	1.632 K	13.733	Normal	10	91	128	Marketing Tools
nvvsvc.exe	816	SYSTEM	00	4.196 K	4.220 K	1.468 K	1.169	Normal	5	10	5	NVIDIA Driver Helper Service, Version 188.80
nvvsvc.exe	1700	SYSTEM	00	10.008 K	10.700 K	3.640 K	6.542	Normal	5	1	0	NVIDIA Driver Helper Service, Version 188.80

Processo = Programa?

- 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 DLL no Windows

Processo Como Uma Máquina Virtual

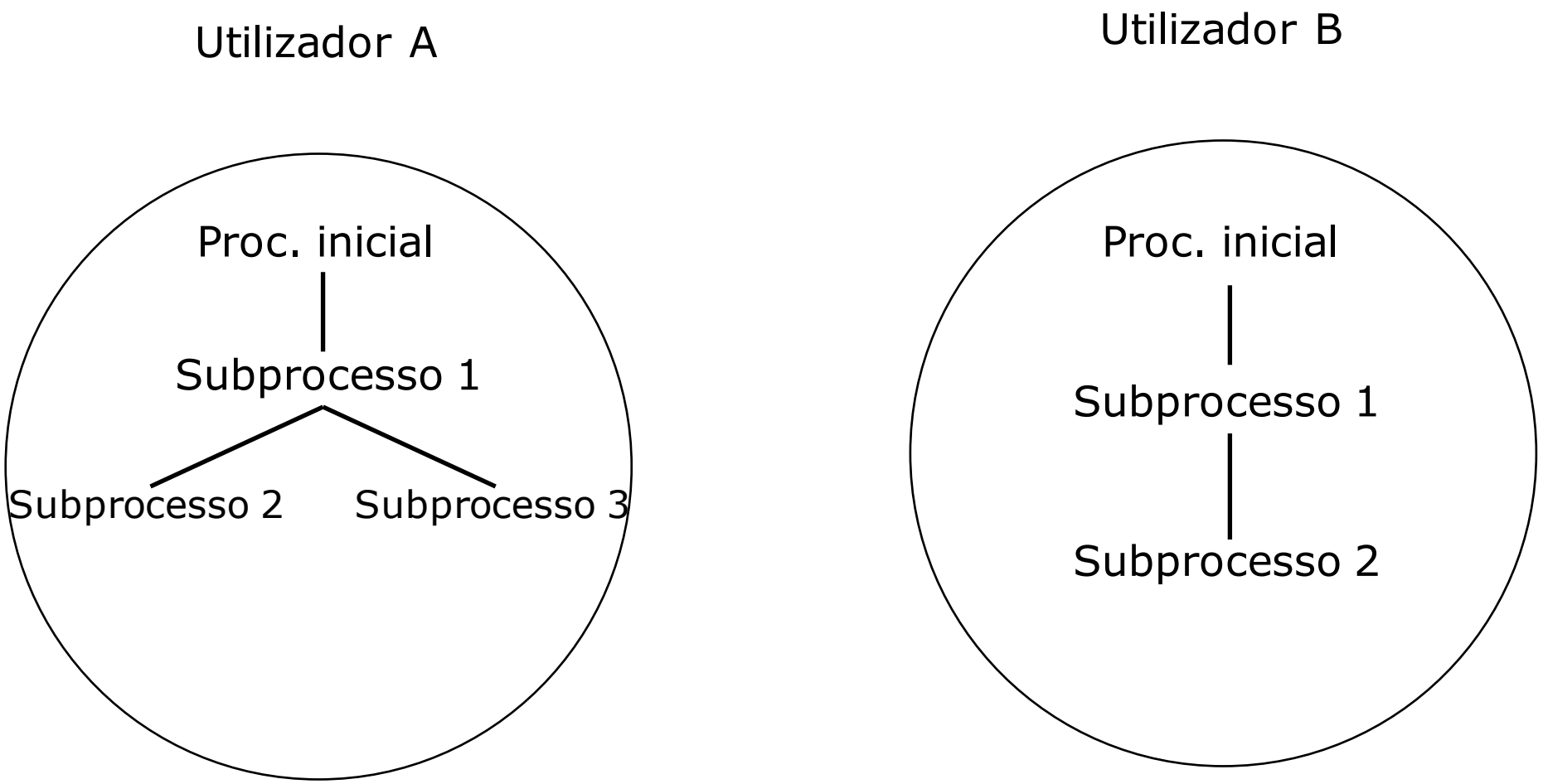


Elementos principais da máquina virtual que o SO disponibiliza aos processos

Processo Como Uma Máquina Virtual

- Tal como uma máquina real, 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):
    - Toda a informação necessária para retomar a execução do processo
    - Memorizado quando o processo é retirado de execução

Hierarquia de Processos



Certas propriedades são herdadas

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

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.

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

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 filho

Programação com processos em Unix

Processos em Unix

- Processos identificados por inteiro (PID)
- Alguns identificadores estão pré atribuídos:
  - Processo 0 é o *swapper* (gestão de memória)
  - Processo 1 init é o de inicialização do sistema

Hierarquia de processos

- Processos relacionam-se de forma hierárquica
- Novo processo herda grande parte do contexto do processo pai
- Quando o processo pai termina os subprocessos continuam a executar-se
    - São adoptados pelo processo de inicialização (pid = 1)

Criação de um Processo

id = fork()

A função não tem parâmetros, em particular o ficheiro a executar.

Processo filho é uma cópia do pai:

- O espaço de endereçamento é copiado

- Contexto de execução é copiado

Então que atributos diferem entre filho e pai?

Estas cópias são pesadas?
Se acontecessem literalmente, seriam.
Na verdade, a chama a fork é muito rápida.
Veremos mais tarde qual o segredo.

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!
Nunca visto em programação sequencial

Exemplo de fork

```
main() {
     int pid;
     pid = fork();
     if (pid == -1) /* tratar o erro */
     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

E se a main terminar com return em vez de exit?

- Até agora, nunca chamámos exit para terminar programas
- Terminação de programa feita usando return (int) na função main do programa
- Qual a diferença?
- Nenhuma, pois o compilador assegura que return da main resulta em chamada a exit!

```
main_aux(argc, argv) {
    int s = main(argc, argv);
    exit(s);
}
```

Função main do programador

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

Como usar o estado de terminação

man wait

[...]

If status is not NULL, wait() and waitpid() store status information in the int to which it points. This integer can be inspected with the following macros (which take the integer itself as an argument, not a pointer to it, as is done in wait() and waitpid()!):

WIFEXITED(status)

returns true if the child terminated normally, that is, by calling exit(3) or _exit(2), or by returning from main().

WEXITSTATUS(status)

returns the exit status of the child. This consists of the least significant 8 bits of the status argument that the child specified in a call to exit(3) or _exit(2) or as the argument for a return statement in main(). This macro should only be employed if WIFEXITED returned true.

WIFSIGNALED(status)

returns true if the child process was terminated by a signal.

WTERMSIG(status)

returns the number of the signal that caused the child process to terminate. This macro should only be employed if WIFSIGNALED returned true.

WCOREDUMP(status)

returns true if the child produced a core dump. This macro should only be employed if WIFSIGNALED returned true. This macro is not specified in POSIX.1-2001 and is not available on some UNIX implementations (e.g., AIX, SunOS). Only use this enclosed in #ifdef WCOREDUMP ... #endif.

[...]

Processo filho nem sempre termina normalmente (com exit)!

Quando termina com exit, inteiro retornado deve ser obtido usando esta macro

Há várias razões para terminação sem 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);
    }
}
```

Exit elimina todo o estado do processo?

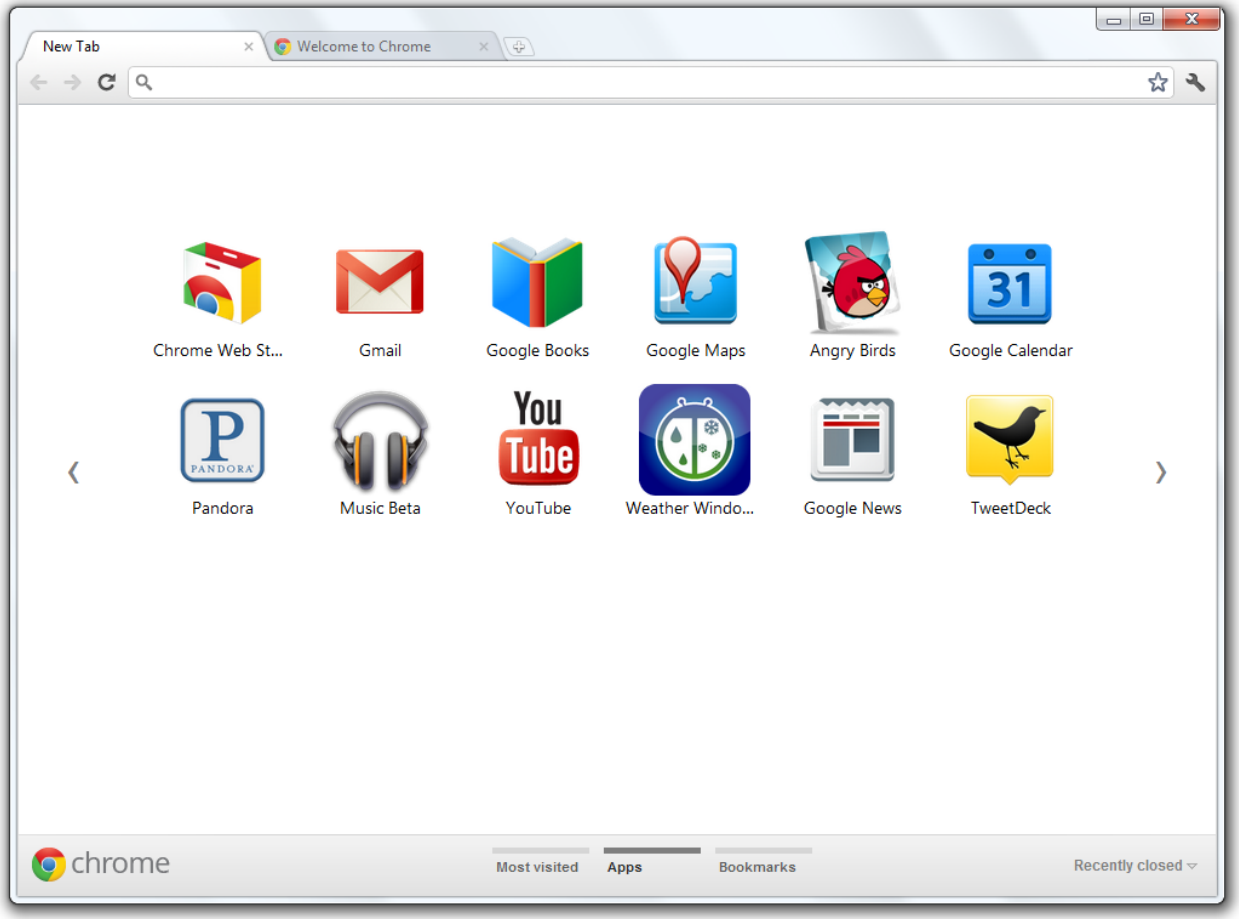
- São mantidos os atributos necessários para quando o pai chamar *wait*:
  - Pid do processo terminado e do seu processo pai
  - Status da terminação
- Entre exit e wait, processo diz-se *zombie*
- Só depois de wait o processo é totalmente esquecido

O minimalismo da função fork

- O fork apenas permite lançar processo com o mesmo código
  - Prós e contras?

Exemplo do uso do fork: Chrome

- No browser Chrome, criar um novo separador causa chamada a fork
- Processo filho usado para carregar e executar scripts dos sites abertos nesse separador
- Porquê?



Como ter filho a executar programa diferente?

```
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

NOTA:
*** execl() e execv() são "front-ends" mais simples para execve() que é a função principal com mais parâmetros**

Como ter filho a executar programa diferente?

- A função exec **substitui** o espaço de endereçamento do processo onde é invocada por aquele contido num ficheiro executável
  - Programa substituído pelo programa no ficheiro
  - Dados (heap) substituídos pelos dados iniciais do ficheiro
  - Pilha é esvaziada
  - *Instruction pointer* aponta para a 1ª instrução do main do novo programa
  - Parâmetros de entrada da nova função main: são os mesmos que foram passados como argumento à função exec
- Quando exec tem sucesso, não há retorno
  - Porquê?
- Restante contexto de execução mantém-se intacto
  - Exemplos de atributos que se mantêm?
  - Como fazer caso se queira que alguns desses atributos não sejam herdados pelo filho?

Exemplo de Exec

```
main ()
{
    int pid;

    pid = fork ();
    if (pid == 0) {
       // which who … or which <some_command>
        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

Um exemplo completo: Shell

- O shell constitui um bom exemplo da utilização de fork e exec (esqueleto muito simplificado)

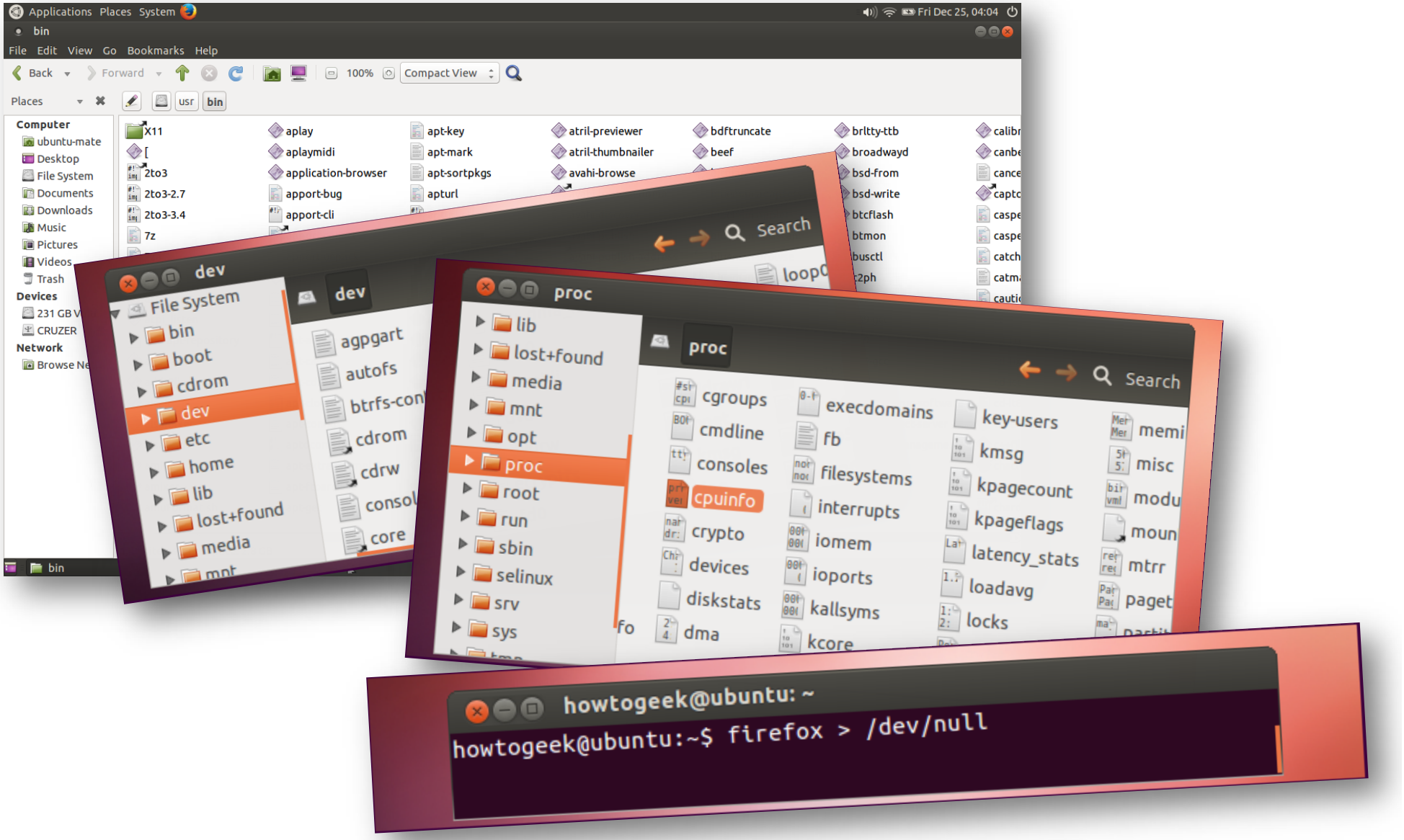
```
while (TRUE){
    prompt();
    read_command (command, params);

    pid = fork ();
    if (pid < 0) {
        printf ("Unable to fork");
         continue;
    }
    if (pid !=0) {
        wait(&status)
    } else{
        execv (command, params);
    }
}
```

Programação com processos: acesso a objetos geridos pelo SO

Sistemas Operativos

2018 / 2019

“Everything is a file”



O princípio *"Everything is a file"*

- Objetos que o SO gere são acessíveis aos processos através de *descritores de ficheiro*
  - Ficheiros, diretorias, dispositivos lógicos, canais de comunicação, etc.
- Vantagens para os utilizadores/programadores
  - Modelo de programação comum
  - Modelo de segurança comum
- Um dos princípios chave do Unix
  - Seguido por muitos SOs modernos
  - Algumas excepções (até no Unix)

Modelo de programação: A API do sistema de ficheiros (Revisão de IAED)

Abordagem 1: Trabalhar com ficheiros usando as funções da stdio

Operações sobre ficheiros

- Até este momento fizemos sempre leituras do stdin e escrevemos sempre para o stdout. Vamos ver agora como realizar estas operações so

```
FILE *fp;

fp=fopen(”tests.txt", "r");
```

Ponteiro para estrutura que representa o ficheiro aberto

Modo de abertura do ficheiro. Neste caso estamos a abrir o ficheiro em modo de leitura

Operações sobre ficheiros

- Até este momento fizemos sempre leituras do stdin e escrevemos sempre para o stdout. Vamos ver agora como realizar estas operações sobre ficheiros.

Modos de abertura

r – abre para leitura (read)

w – abre um ficheiro vazio para escrita (o ficheiro não precisa de existir)

a – abre para acrescentar no fim (“append” ; ficheiro não precisa de existir)

r+ – abre para escrita e leitura; começa no início; o ficheiro tem de existir

w+ – abre para escrita e leitura (tal como o “w” ignora qualquer ficheiro que exista com o mesmo nome, criando um novo ficheiro)

a+ – abre para escrita e leitura (output é sempre colocado no fim)

…mas há mais

Exemplo

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

int main()
{
  FILE *fp;

  fp = fopen("teste.txt", "r");
  if (fp == NULL) {
    printf("teste.txt: No such file or directory\n");
    exit(1);
  }

  return 0;
}
```

Se não conseguir abrir, fp fica igual a NULL

Exemplo

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

int main()
{
  FILE *fp;

  fp = fopen("teste.txt", "r");
  if (fp == NULL) {
    perror("teste.txt");
    exit(1);
  }

  return 0;
}
```

Escreve a mesma mensagem de erro.

perror() escreve no "standard error" (stderr) a descrição do último erro encontrado na chamada a um sistema ou biblioteca.

Exemplo 2

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

int main()
{
  FILE *fp;

  fp = fopen("teste.txt", "r");
  if (fp == NULL) {
    perror("teste.txt");
    exit(1);
  }

  fclose(fp);

  return 0;
}
```

Fecha o ficheiro

Exemplo 3

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

int main()
{
  FILE *fp;

  fp = fopen("teste.txt", "w");
  if (fp == NULL) {
    perror("teste.txt");
    exit(1);
  }

  fprintf(fp, "Hi file!\n");

  fclose(fp);

  return 0;
}
```

Escreve para um ficheiro

Exemplo 3

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

int main()
{
  FILE *fp;

  fp = fopen("teste.txt", "w");
  if (fp == NULL) {
    perror("teste.txt");
    exit(1);
  }

  fputs("Hi file!", fp);

  fclose(fp);

  return 0;
}
```

Escreve para um ficheiro (alternativa)

Exemplo 4

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

int main()
{
  FILE *myfile; int i;
  float mydata[100];
  myfile = fopen("info.dat", "r");
  if (myfile== NULL) {
    perror("info.dat");
    exit(1);
  }
  for (i=0;i<100;i++)
     fscanf(myfile,"%f",&mydata[i]);

  fclose(myfile);
  return 0;
}
```

Lê um conjunto de 100 floats guardados num ficheiro

Exemplo 5

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

int main()
{
  FILE *myfile; int i;
  myfile = fopen("info.dat", "a");

  for (i=0;i<100;i++)
     fprintf(myfile,"%d\n",i);

  fclose(myfile);
  return 0;
}
```

Adiciona um conjunto de 100 inteiros ao fim de um ficheiro

Exercício: Escrever matriz para ficheiro

- Pegar no grid.c
- Criar nova função, grid_print_to_file que imprime a grelha em ficheiro passado como argumento
- Chamar esta função sobre ficheiro "out.txt" aberto no final da função *maze_checkPaths* em maze.c

void grid_print_to_file (grid_t* gridPtr, FILE *f)

O cursor

- Para qualquer ficheiro aberto, é mantido um cursor
- O cursor avança automaticamente com cada byte lido ou escrito

- Para sabermos em que posição estamos, usar função ***ftell***

```
long ftell(FILE *stream);
```

- Para repor o cursor noutra posição, usar função ***fseek***
  - Por exemplo, colocar cursor no início ou final do ficheiro

```
int fseek(FILE *stream, long offset,
int whence);
```

Escritas são imediatamente persistentes?

- Após escrita em ficheiro, essa escrita está garantidamente persistente no disco?
  - Nem sempre!
  - Para optimizar o desempenho, escritas são propagadas para disco tardiamente.
- Função ***fflush*** permite ao programa forçar que escritas feitas até agora sejam persistidas em disco
  - Função só retorna quando houver essa garantia
  - Função demorada, usar apenas quando necessário

```
int fflush(FILE *stream);
```

Modelo de programação: A API do sistema de ficheiros (Revisão de IAED)

Abordagem 2: Trabalhar com ficheiros usando as funções da API do sistema de ficheiros do Unix

O que ganho/perco?

Prós:

- Em geral, são funções de mais baixo nível, logo permitem maior controlo
- Algumas operações sobre ficheiros só estão disponíveis através desta API

Contras:

- Normalmente, programa que usa stdio é mais simples e optimizado
  - Discutiremos mais à frente em SO porque é que stdio é mais optimizado

Sistema de Ficheiros do Unix

Operações	Genéricas	Linux
Simples	Fd := Abrir (Nome, Modo) Fd := Criar (Nome, Protecção)	int open(const char *path, int flags, mode_t mode)
	Fechar (Fd)	int close(int fd)
Ficheiros Abertos	Ler (Fd, Tampão, Bytes)	int read(int fd, void *buffer, size_t count)
	Escrever (Fd, Tampão, Bytes)	int write(int fd, void *buffer, size_t count)
	Posicionar (Fd, Posição)	int lseek(int fd, off_t offset, int origin)
Complexas	Criar link (Origem, Destino)	int symlink(const char *oldpath, const char *newpath) int link(const char *oldpath, const char *newpath)
	Mover (Origem, Destino)	int rename(const char *oldpath, const char *newpath)
	Apagar link (Nome)	int unlink(const char *path)
		int dup(int fd), int dup2(int oldfd,int newfd)
	LerAtributos (Nome, Tampão)	int stat(const char *path, struct stat *buffer)
	EscreverAtributos (Nome, Atributos)	int fcntl(int fd,int cmd,struct flock *buffer) int chown(const char *path, uid_t uid, gid_t gid) int chmod(const char *path, mode_t mode)
Ficheiros em memória	MapearFicheiro(Fd,pos,endereço,dim)	void *mmap(void *addr, size_t len, int prot, int flags, int fd, off_t offset)
	DesMapearFicheiro(endereço,dim)	int munmap(void *addr, size_t len)
Directórios	ListaDir (Nome, Tampão)	int readdir(int fd, struct dirent *buffer, unsigned int count)
	MudaDir (Nome)	int chdir(const char *path)
	CriaDir (Nome, Protecção)	int mkdir(const char *path, mode_t mode)
	RemoveDir(Nome)	int rmdir(const char *path)
Sistemas de Ficheiros	Montar (Directório, Dispositivo)	int mount(const char *device, const char *path, const char *fstype, unsigned long flags, const void *data)
	Desmontar (Directório)	int umount(const char *path)

Sistema de Ficheiros do Unix: Depois desta aula

- Estudar as *man pages* destas funções
- Em particular, as funções *unlink* e *rename* serão certamente úteis no 3º exercício do projeto

- Analisaremos o funcionamento interno destas funções dentro de algumas semanas nas teóricas de SO

Modelo de segurança

Modelo de Segurança

- Processo em execução associado a um **Utilizador**
  - Cada utilizador identificado por **User IDentifier (UID)**
  - Processo executa operaçoes em nome desse utilizador
- Para facilitar a partilha, o utilizador pertence a um ou mais grupos de utilizadores
  - Cada grupo identificado por um Group Identifier (GID)

Controlo dos Direitos de Acesso

1. Processo pede para executar operação sobre objecto gerido pelo núcleo
2. Núcleo valida se o UID/GID do processo tem direitos para executar aquela operaçao sobre aquele objeto
3. Se sim, núcleo executa operação; se não, retorna erro

Controlo dos Direitos de Acesso

- Conceptualmente, a autorização baseia-se numa Matriz de Direitos de Acesso

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

- Colunas designam-se Listas de Direitos de Acesso (ACL)
- Linhas designam-se por Capacidades

Autenticação

- Um processo tem associados dois identificadores:
  - o número de utilizador, UID (*user identification*)
  - o número de grupo, GID (*group identification*)
- *superuser* (ou *root*) tem UID especial (zero)
  - Autorizado a realizar quaisquer operações sobre os recursos lógicos
  - superuser = execução em modo núcleo?

Autenticação

- Atribuídos quando o utilizador se autentica (*log in*) perante o sistema
  - Os UID e GID são obtidos do ficheiro /etc/passwd no momento do login
- O UID e o GID são herdados pelos processos filhos

Autenticação de processos

- Cada processo corre em nome de um utilizador (UID)/grupo (GID)
- Na verdade, há:
  - Real UID e real GID
    - » Normalmente nunca mudam
  - Effective UID e effective GID
    - » Podem mudar temporariamente
  - E também há o saved UID/GID (fora da matéria)
- Quando o processo faz chamadas de sistema, o núcleo consulta o seu EUID/EGID para determinar se tem permissão

Herança de UID/GID

- UID e GID são herdados pelo processos filho
  - Real e effective
- Se tudo começa num primeiro processo a correr em nome do root, como pode haver processos (filhos) a correr em nome de outros utilizadores?

Mudar de UID/GID

- Há diferentes vias para um processo mudar de UID/GID
- Primeira via:
  - Funções setuid(int) e setgid(int)
- Mudam effective UID/GID
- Restrições caso seja chamada por processo não-root (ver man pages)

Mudar de UID/GID

- Segunda via:

–Executável de outro UID/GID com bit setUID/setGID ativo

–Processo que execute esse executável (chamando exec*) adquire EUID/EGID do dono do ficheiro

- Isto é útil?

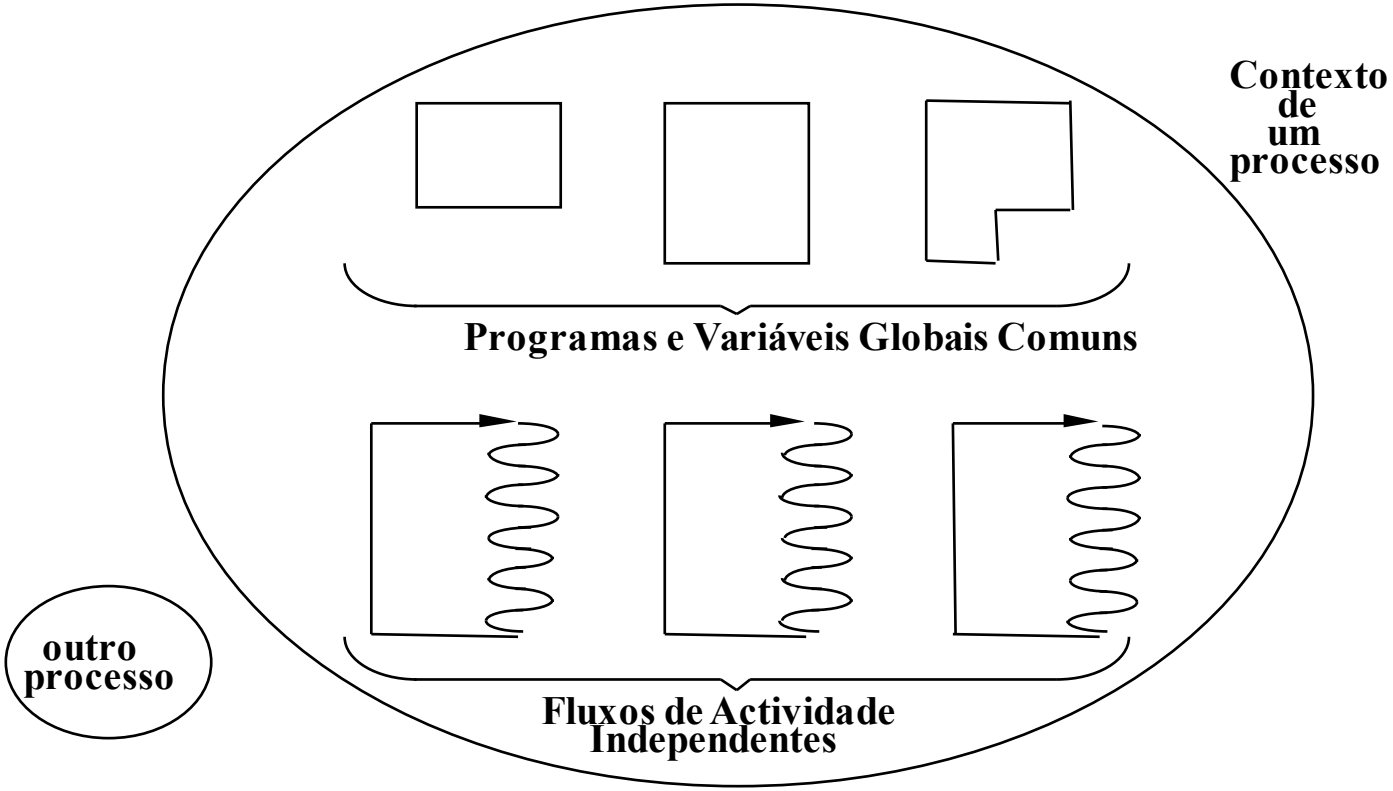
Introdução à programação com tarefas (*threads*)

Sistemas Operativos

2018 / 2019

Tarefas

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



Paralelismo com múltiplos processos vs. múltiplas tarefas

Quando faz sentido paralelizar um projeto com cada alternativa?

O que é partilhado entre tarefas do mesmo programa

- O código
- Amontoado (heap)
  - Variáveis globais
  - Variáveis dinamicamente alocadas
- Atributos do processo

  (Veremos mais tarde)

O que não é partilhado entre tarefas do mesmo processo

- Pilha (stack)
  - Mas atenção: não há isolamento entre pilhas!
  - *Bugs* podem fazer com que uma tarefa aceda à pilha de outra tarefa
- Estado dos registos do processador
  - Incluindo *instruction pointer*
- Atributos específicos da tarefa
  - Thread id (tid)
  - Etc (veremos mais tarde)

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)

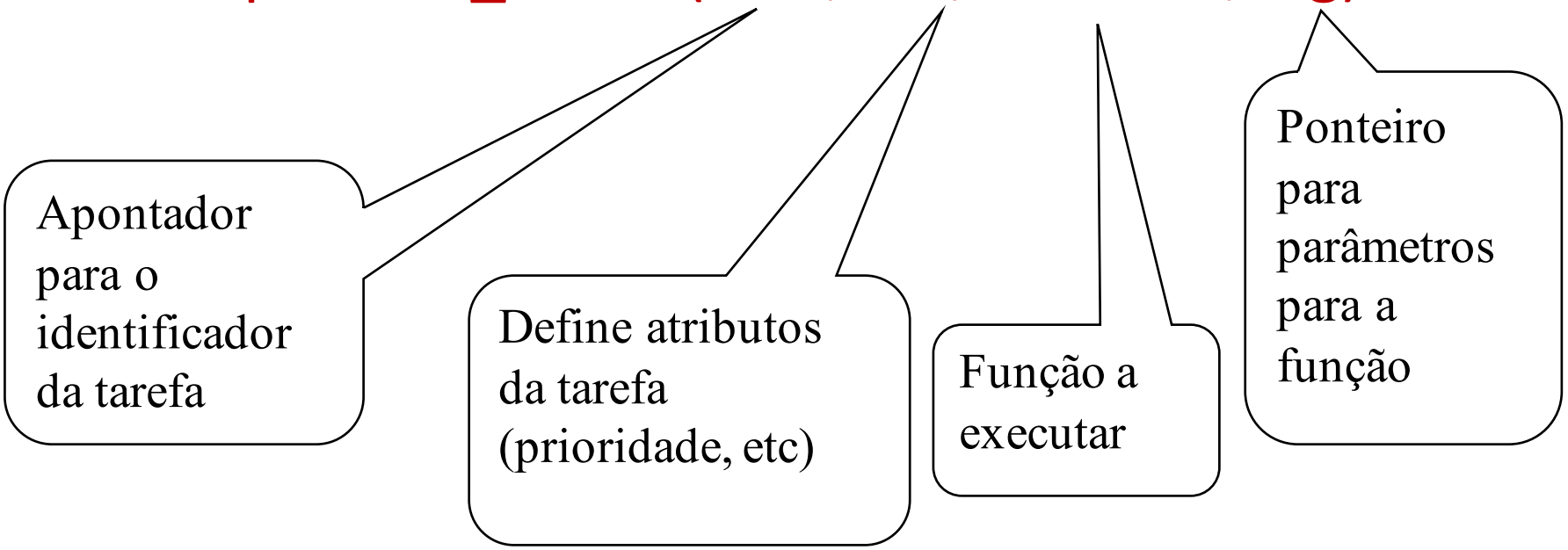
Bloqueia a tarefa à espera da terminação de outra tarefa ou da tarefa referenciada no parâmetro Idtarefa

Programação de processos multi-tarefa em Unix/etc

Interface POSIX

Interface POSIX: criar tarefa

pthread_create(&tid, attr, function, arg)



Interface POSIX: terminação de tarefa

pthread_exit(void *value_ptr)

- Tarefa chamadora termina
- Retorna ponteiro para resultados

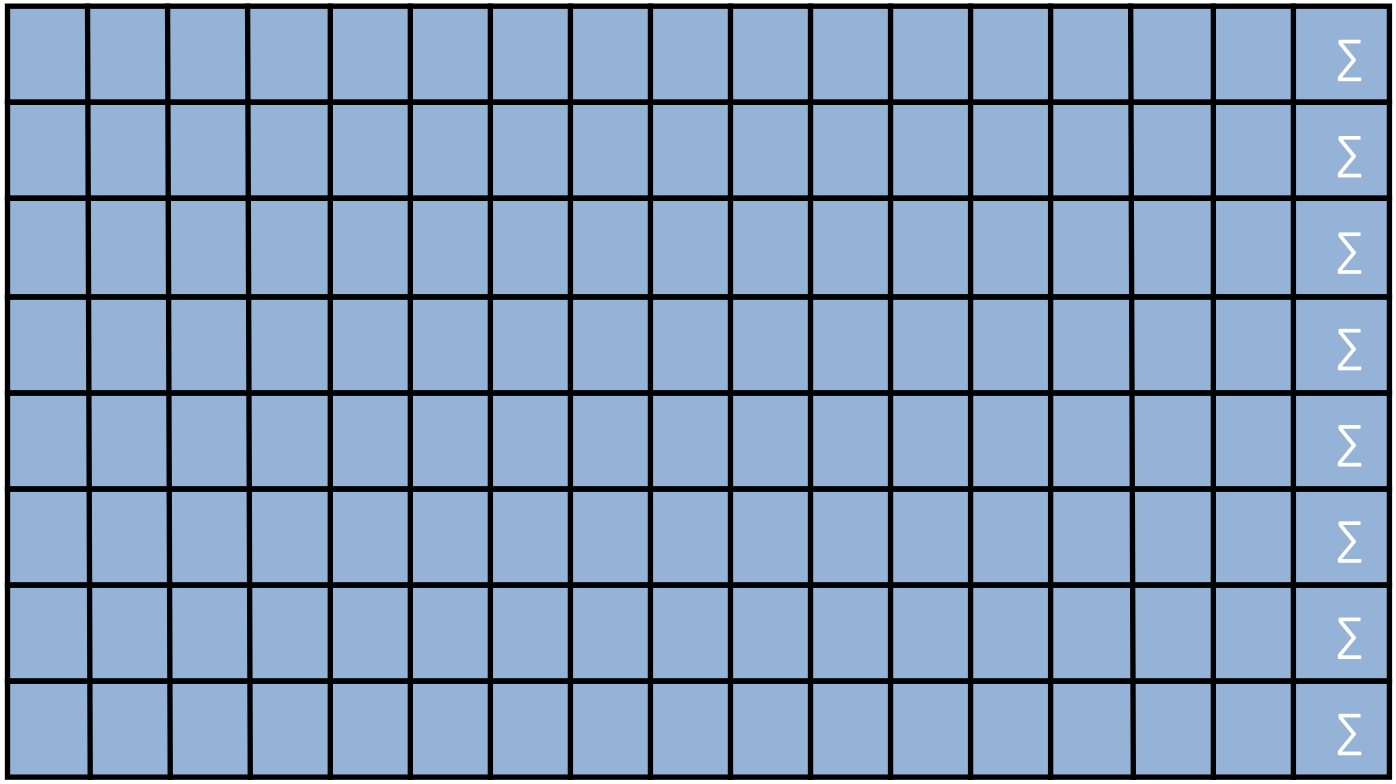
int pthread_join(pthread_t thread,
void**value_ptr)

- Tarefa chamadora espera até a tarefa indicada ter terminado
- O ponteiro retornado pela tarefa terminada é colocado em (*value_ptr)

Regra de ouro

- O núcleo oferece a ilusão de uma máquina com número infinito de processadores, sendo que cada tarefa corre no seu processador
- No entanto, as velocidades de cada processador virtual podem ser diferentes e não podem ser previstas
  - Porquê?
  - Consequências para o programador?

Exemplo: somar linhas de matriz

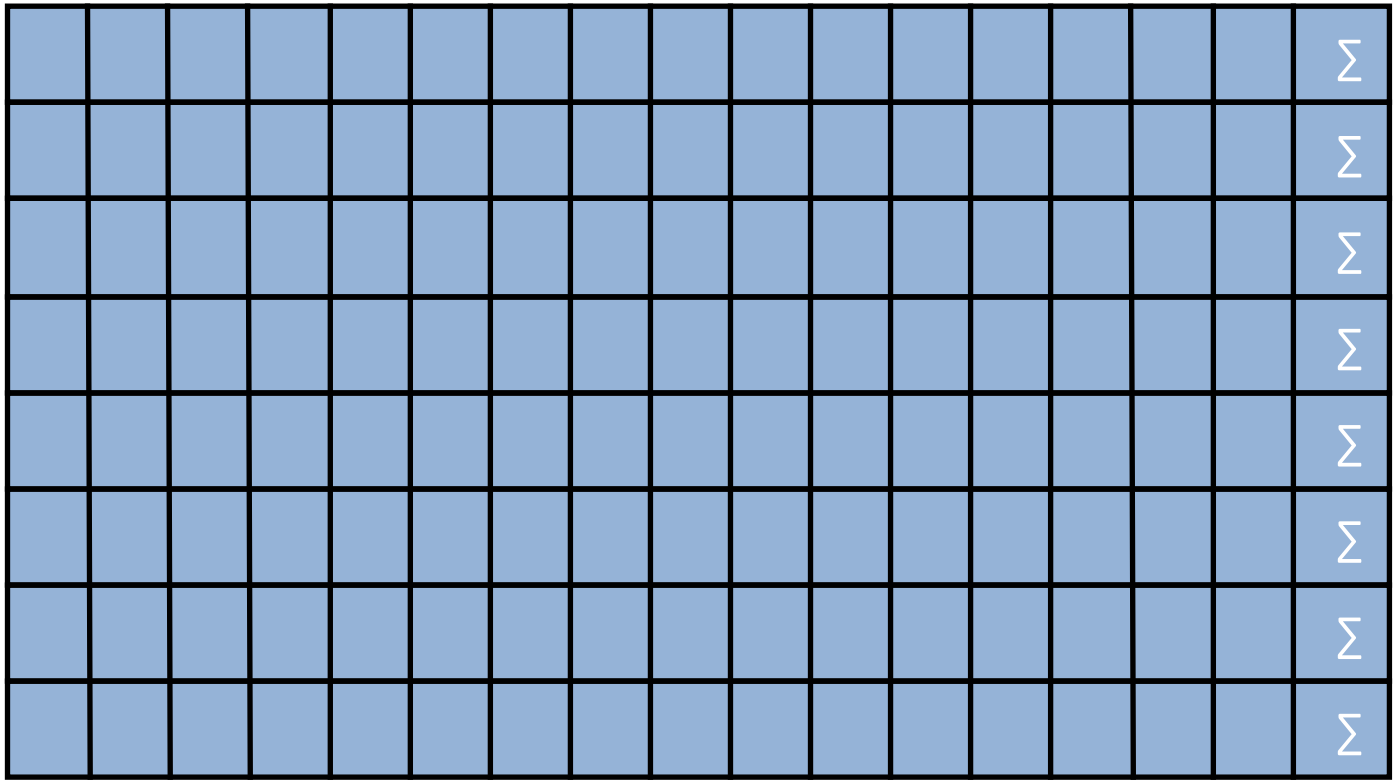


Solução sequencial

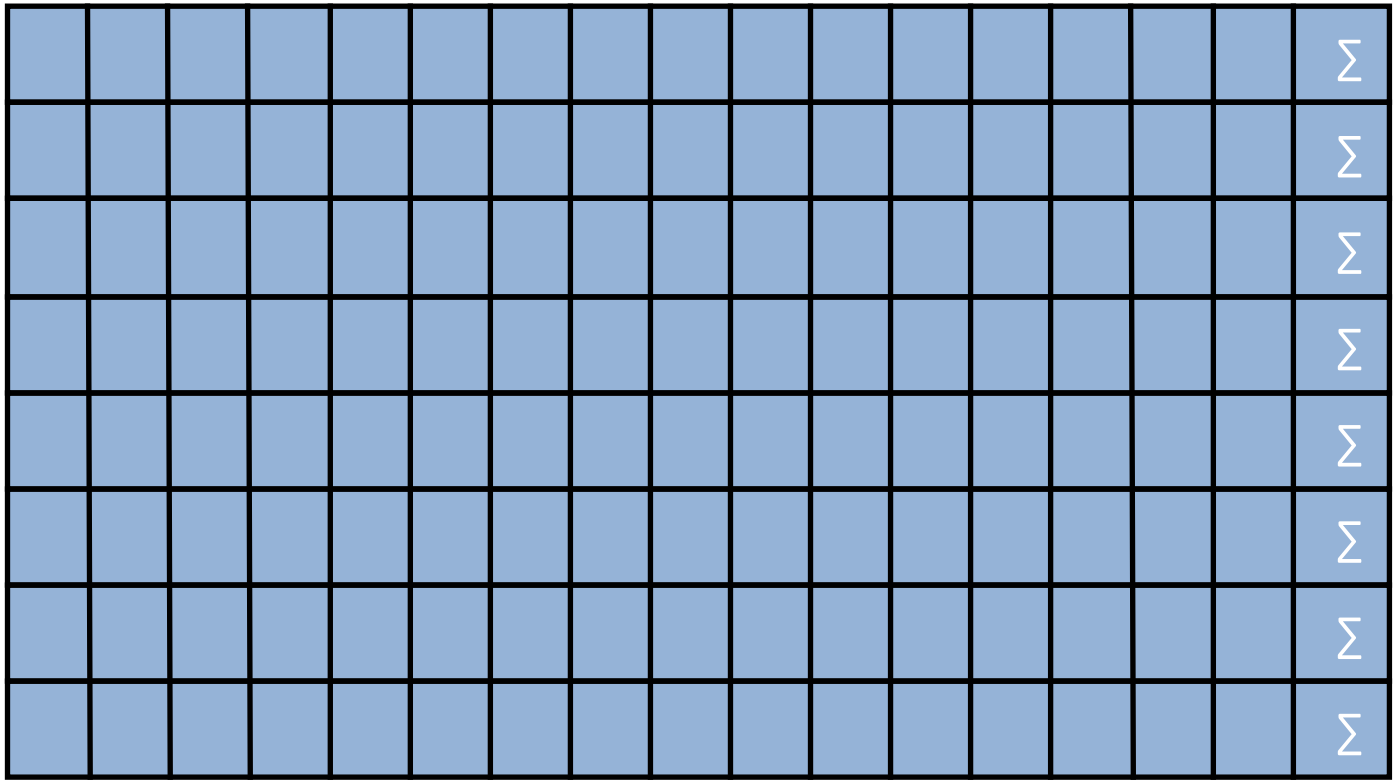
```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>
#define N 5
#define TAMANHO 10

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;

  inicializaMatriz(buffer(N, TAMANHO);

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


  imprimeResultados(buffer);

  exit(0);
}
```

Execução sequencial



Execução em N tarefas paralelas



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)

```c
#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;
}
```

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

 inicializaMatriz(buffer(N, TAMANHO);

 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);
}
```

Exemplo: obter valor de retorno pela pthread_join

```
#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.*/
  int* ret=(int*) malloc(sizeof(int));
  *ret=soma;
  return ret;
}
```

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

 inicializaMatriz(buffer(N, TAMANHO);

 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], &results[i]);
   printf("A tarefa %d devolveu %d\n",i,*results[i]);
 }
 printf ("Terminaram todas as threads\n");

 imprimeResultados(buffer);

 exit(0);
}
```

Então e se o programa não for embaraçosamente paralelo?

Veremos na próxima aula…

Atenção!
Qual o problema neste programa?

```
void *threadFn(void *arg) {
 MyStruct *s = (MyStruct*) arg;
 printf("Nova thread criada com user %d:%s\n",
        s->userId, s->userId);
 ...
}


int main (void) {

 MyStruct s;

 for (i=0; i< N; i++){

   //Prepara argumentos da próxima thread
   s.userId = i;
   s.userName = getUserName(i);

   //Cria thread, passando-lhe um user novo
   if(pthread_create (&tid[i], 0, threadFn, &s)== 0) {
      printf ("Criada a tarefa %d\n", tid[i]);
   }
   else ...
 }
 ...
}
```