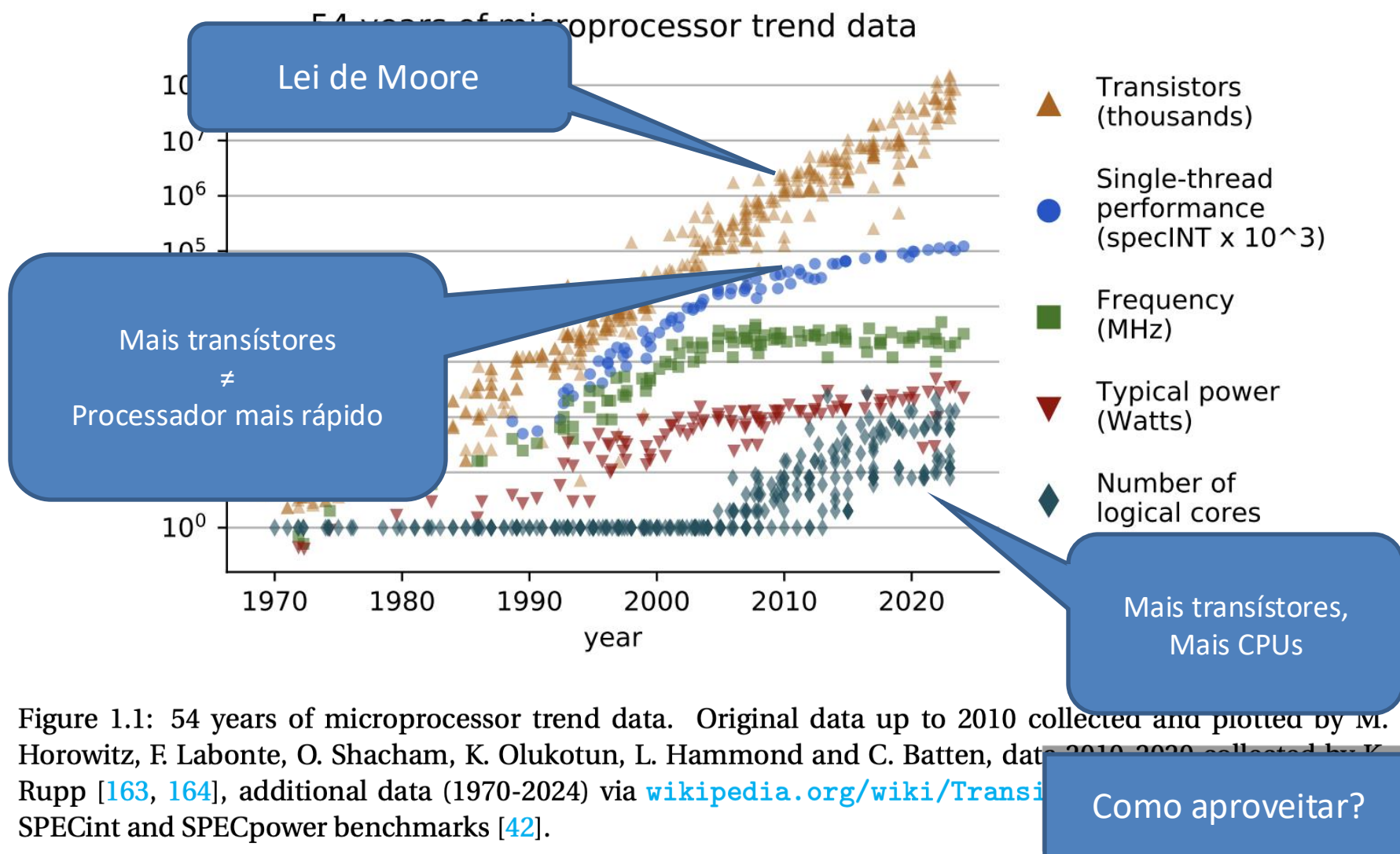
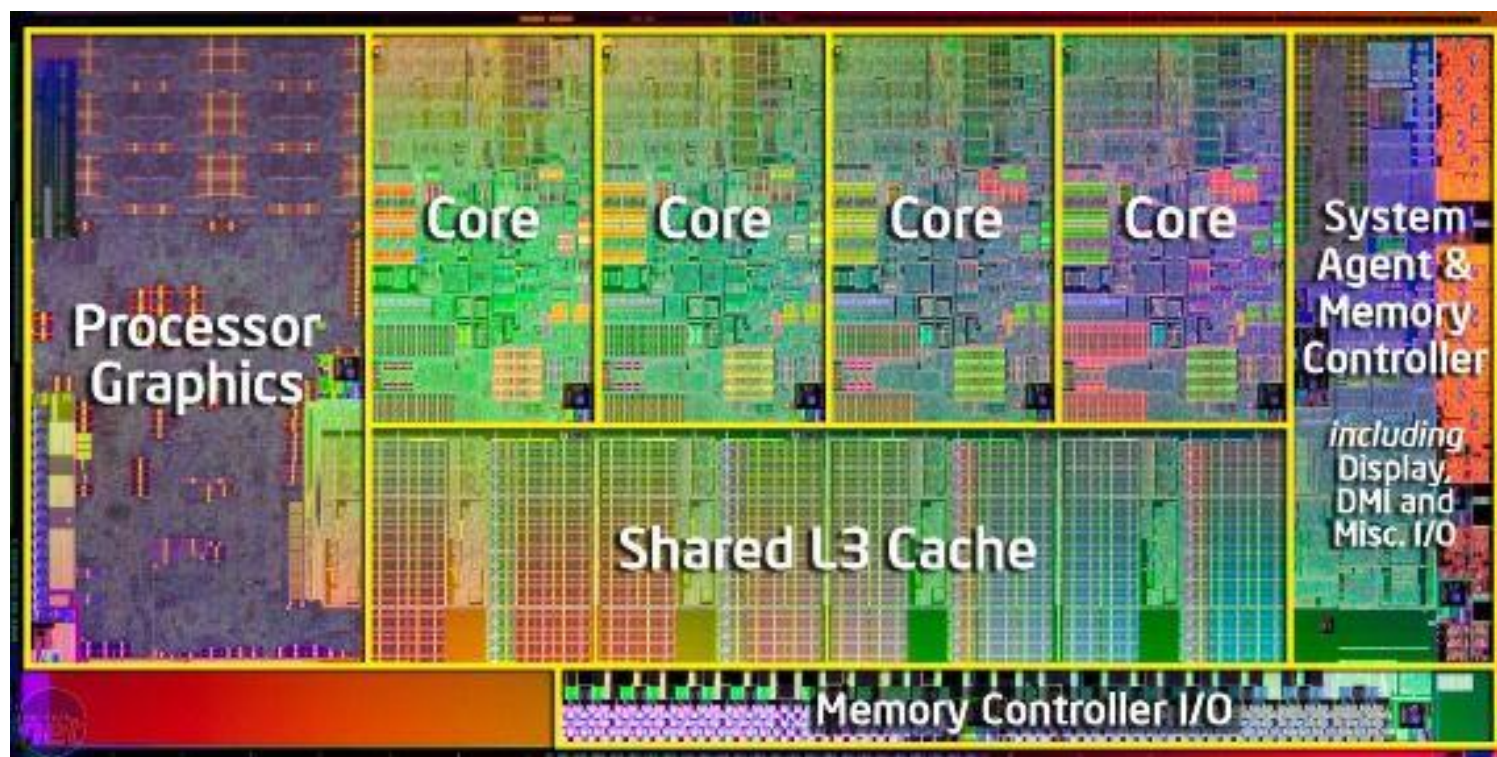


# O fim dos “almoços grátis”



# 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
  - Enquanto um fluxo de execução espera por ação do utilizador, outros podem progredir em fundo

Ou seja, programação paralela faz sentido mesmo em máquinas *single-cpu*!

# Próximas aulas:

## Dois níveis de programação paralela (processos e tarefas)



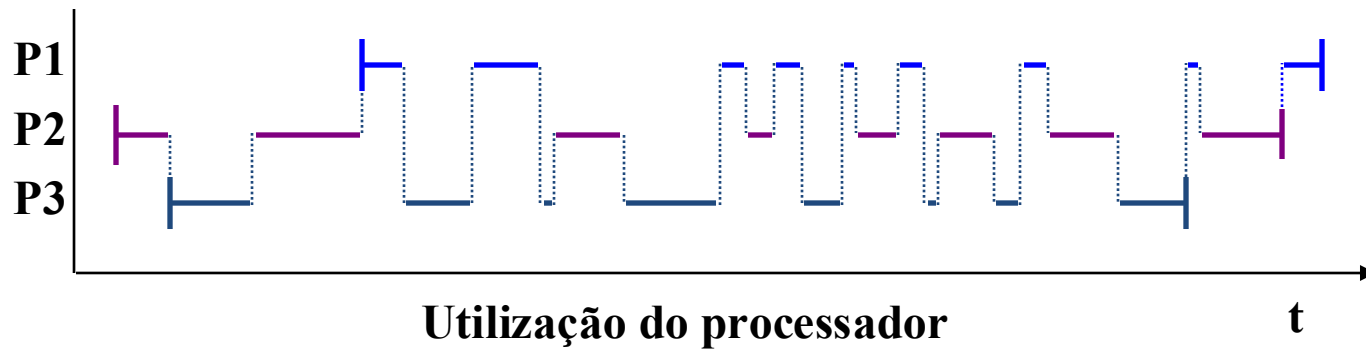
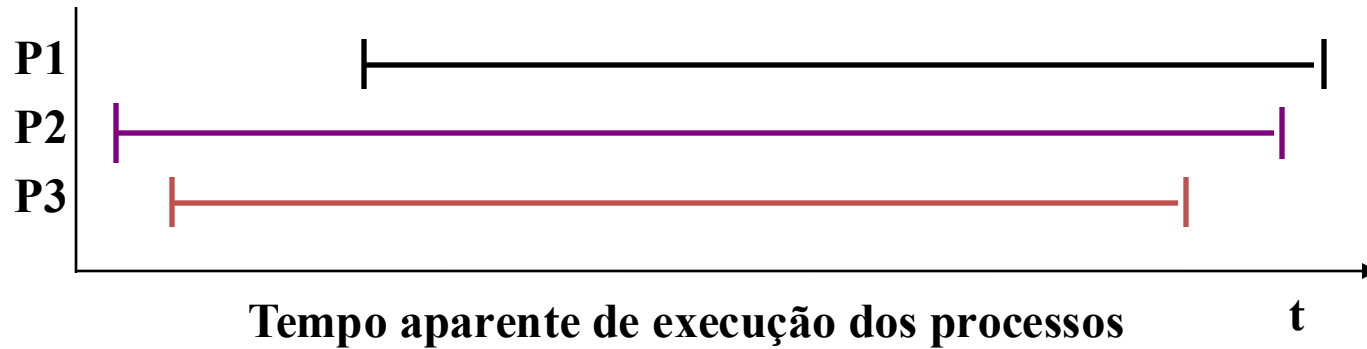
# Introdução à programação com processos

---

# Multiprogramação

- Execução, em paralelo, de múltiplos programas na mesma máquina
- Cada instância de um programa em execução denomina-se um **processo**

# Pseudoconcorrência



# 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



# 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

Windows Task Manager

File Options View Help

Applications Processes Services Performance Networking Users

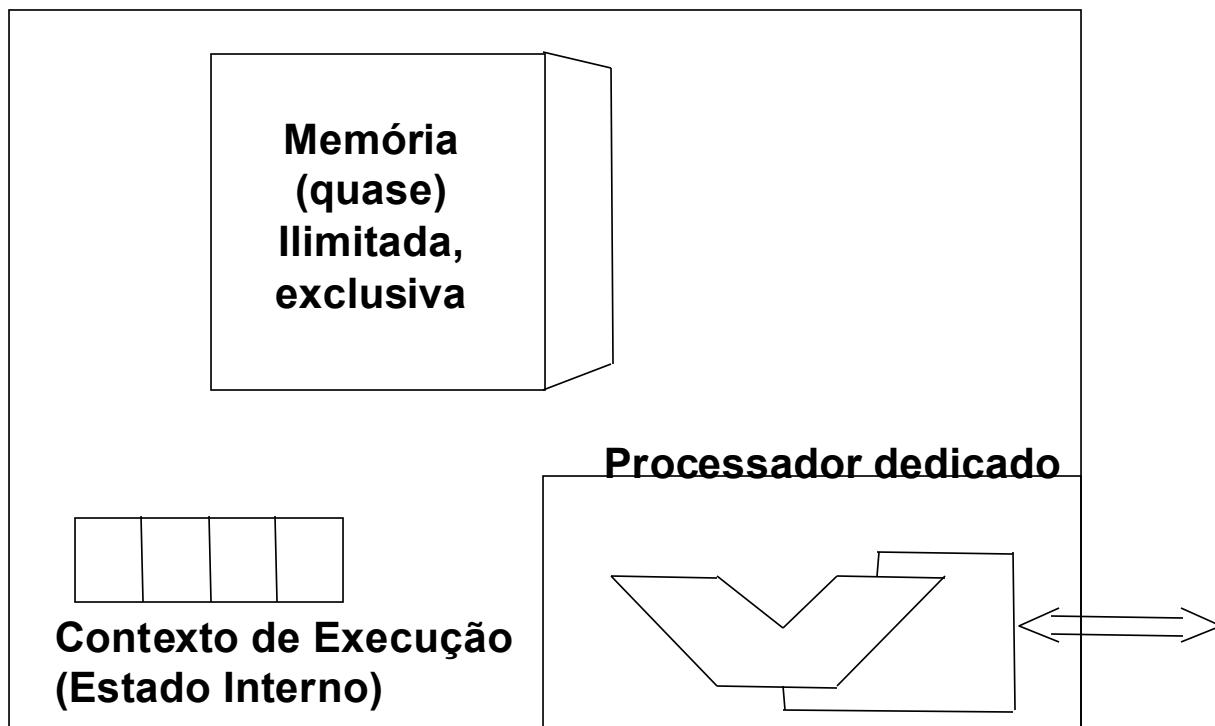
Image Name	PID	User Name	CPU	Working Set (Memory)	Peak Working Set (Memo...	Memory (Private Working Set)	Page Faults	Base Pri	Threads	I/O Reads	I/O Writes	Description
audiodg.exe	3644	LOCAL SERVICE	00	16.852 K	23.796 K	10.660 K	11.380	Normal	6	0	0	Windows Audio Device Graph Isolation
BTTTray.exe	4380	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
dhtml.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
FSHLL32.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
lsn.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
nvwsv.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
nvwsv.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

☒ Show processes from all users

End Process

Processes: 99 CPU Usage: 3% Physical Memory: 17%

# 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

# Modelo: Objecto “Processo”

- Propriedades
  - Identificador
  - Programa
  - Espaço de Endereçamento (codigo, dados, pilha)
  - 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



# 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

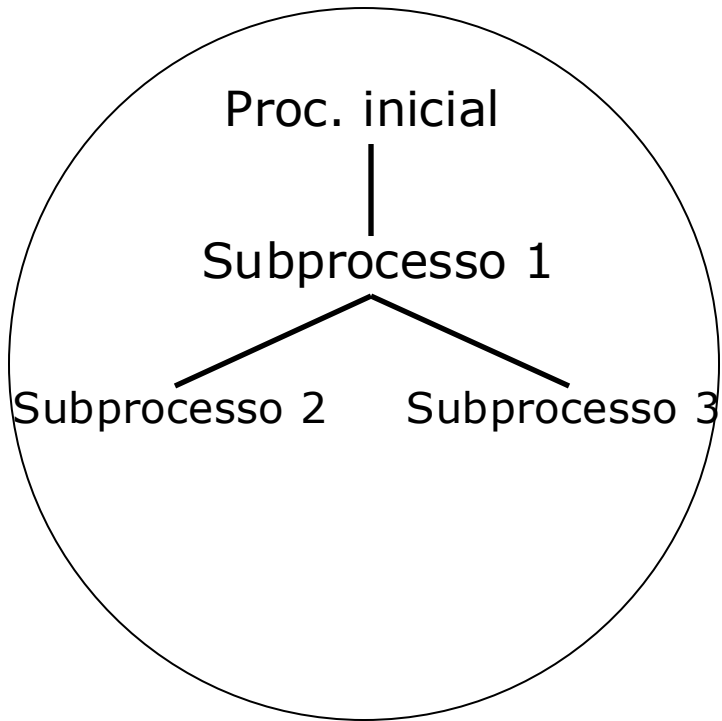
# Hierarquia de processos

- 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)

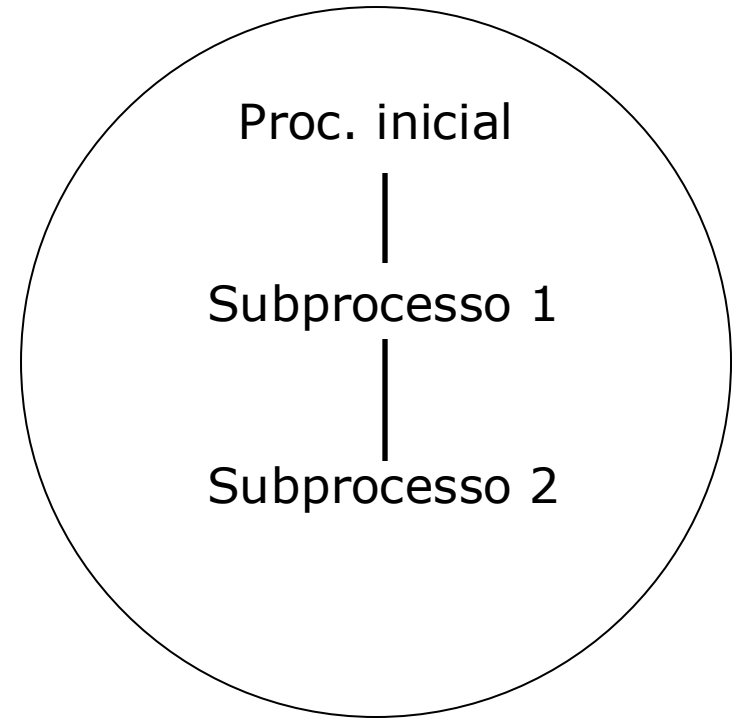


# Hierarquia de Processos

Utilizador A



Utilizador B



Certas propriedades são herdadas

# 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() {
    pid_t 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?

# Como ter o filho a executar um programa diferente?

```
int execl(char* ficheiro, char* arg0, char* arg1,..., argn,0)
```

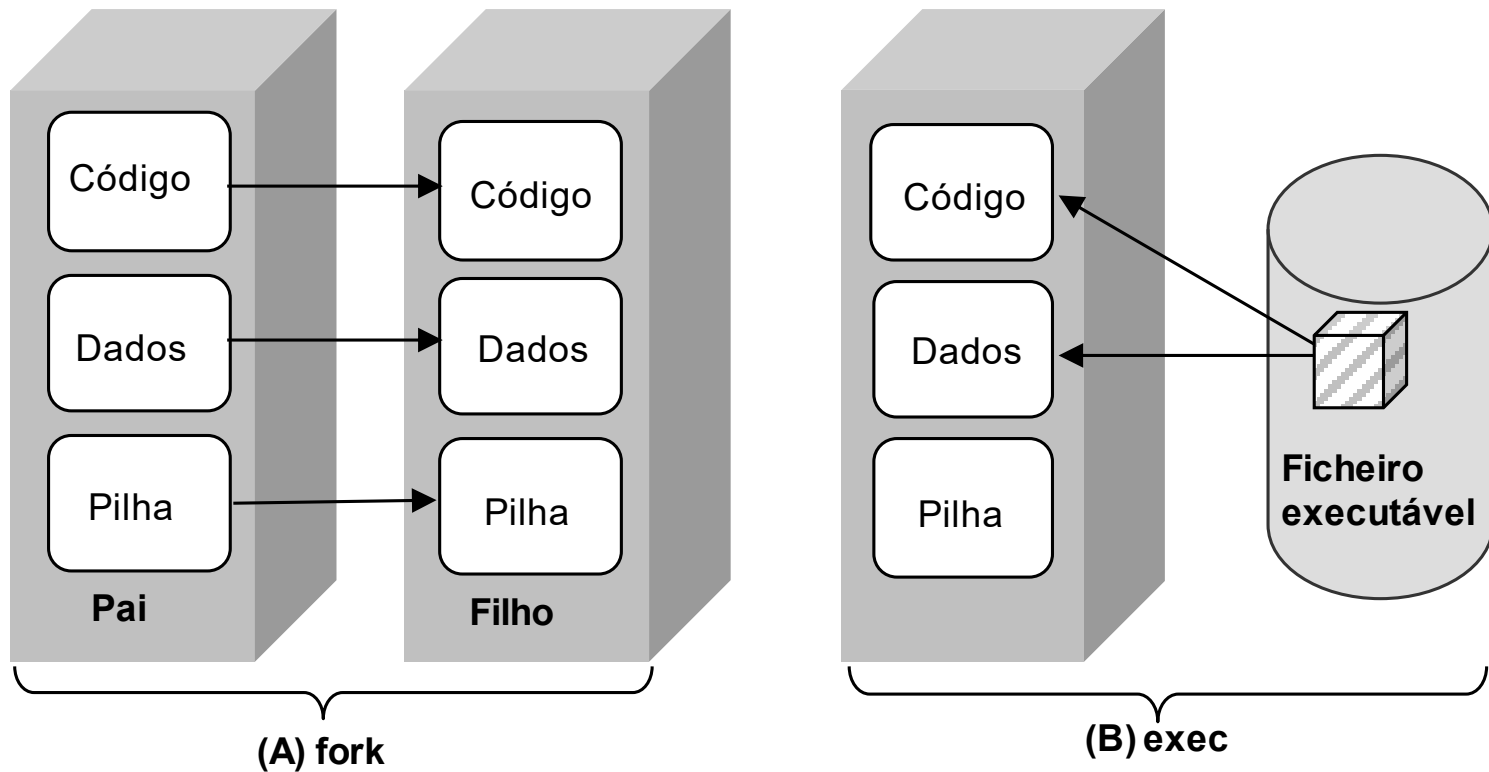
```
int execv(char* ficheiro, char* argv [])
```

**Caminho de acesso ao ficheiro executável**

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

**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**

# Comparação *fork* e *exec*



# 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 substituídos pelos dados iniciais do ficheiro
  - Pilha é esvaziada
  - *Instruction pointer* aponta para a 1ª instrução do *main* do novo programa



# Como ter o filho a executar um programa diferente?

- 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) {
        execl ("/usr/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

# Vamos implementar uma shell?

- Ciclo infinito, em que cada iteração:
  - Imprime mensagem
  - Lê comando
  - Cria novo processo filho
  - Processo filho deve executar outro programa (indicado no comando lido)
  - Entretanto, o processo da shell bloqueia-se até filho terminar
  - Volta à próxima iteração



# 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);
    }
}
```

**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

- Primeira via:
  - funções `setuid(int)` e `setgid(int)`
  - Mudam UID/GID efectivo (*effective*)
  - Restrições caso sejam chamadas por processo sem o privilégio Sistema

# Mudar de UID/GID

- Segunda via:
  - Utilizando o `exec`
  - Ficheiro executável com **com bit setUID/setGID posicionado e outro UID/GID**
  - Processo que execute esse executável (chamando `exec`) adquire UID/GID do dono do ficheiro



# Exemplo de utilização do setUID

```
$ which whoami
/usr/bin/whoami

$ cp /usr/bin/whoami .  #copio o executável para a directoria corrente

$ ls -l whoami
-rwxr-xr-x 1 paolo staff 23264 Dec  5 10:12 whoami

$ ./whoami
paolo

$ sudo chown root whoami

$ ls -l whoami
-rwxr-xr-x 1 root staff 23264 Dec  5 10:12 whoami

$ sudo chmod u+s whoami

$ ls -l whoami
-rwsr-xr-x 1 root staff 23264 Dec  5 10:12 whoami

$ ./whoami
root
```



# Exemplo de utilização do setUID (2)

- No linux, as passwords (criptadas) são armazenadas no ficheiro `/etc/shadow`

> `ls -l /etc/shadow`

```
-rw-r-- 1 root shadow 1824 Oct 18 19:49 /etc/shadow
```

- O commando *passwd* permite que os utilizadores (não superusers) possam alterar as próprias passwords
- Problema: *passwd* precisa aceder `/etc/shadow` para mudar as passwords

> `ls -l /bin/passwd`

```
-rwsr-xr-x 1 root root 68208 May 28 01:37 /bin/passwd
```

- Solução: dono root e user sticky bit ativo!



# Introdução à programação com tarefas (*threads*)

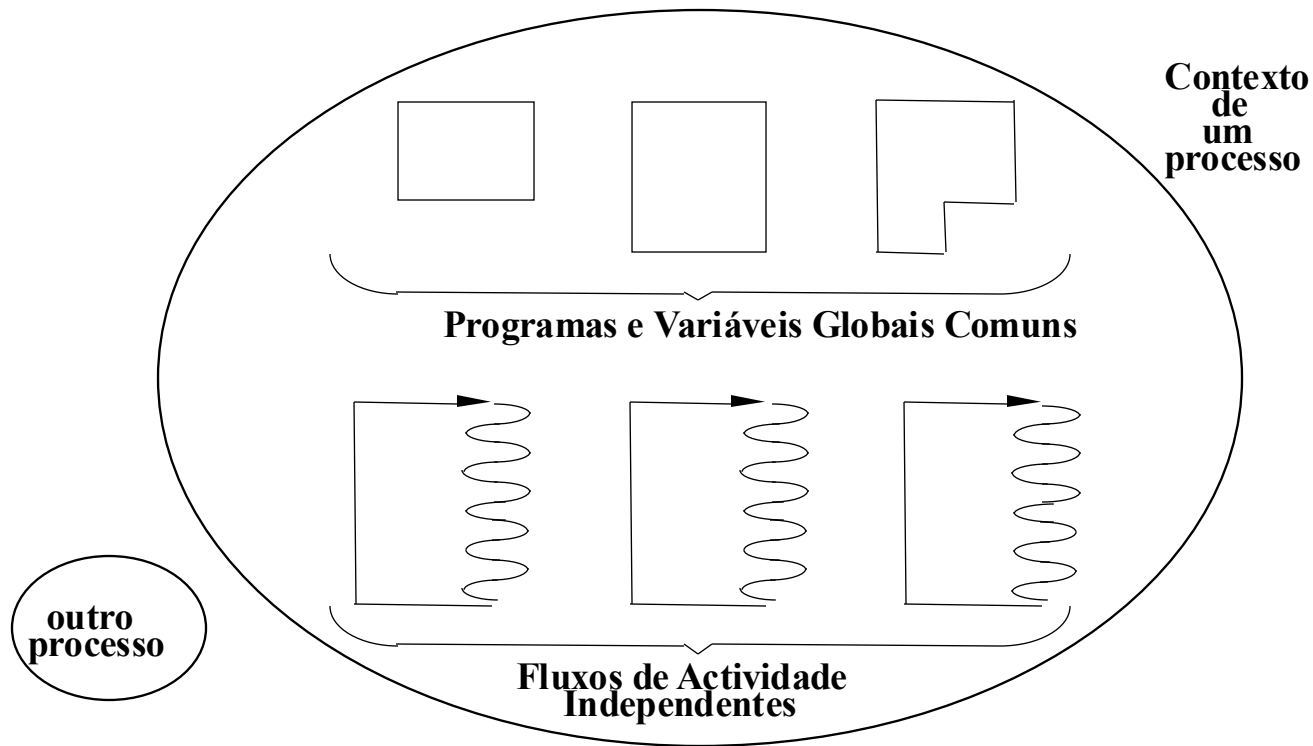
---





# Tarefas

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





# O que é partilhado entre tarefas do mesmo processo

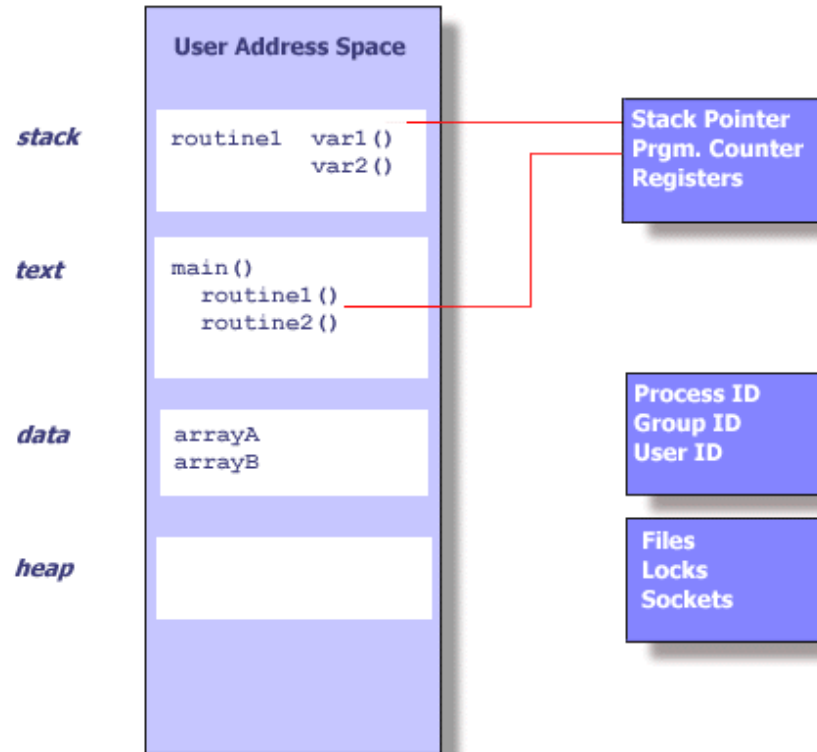
- O código
- Amontoado (heap)
  - Variáveis globais
  - Variáveis dinamicamente alocadas
- Atributos do processo  
(Veremos mais tarde)



# O que não é partilhado entre tarefas do mesmo processo

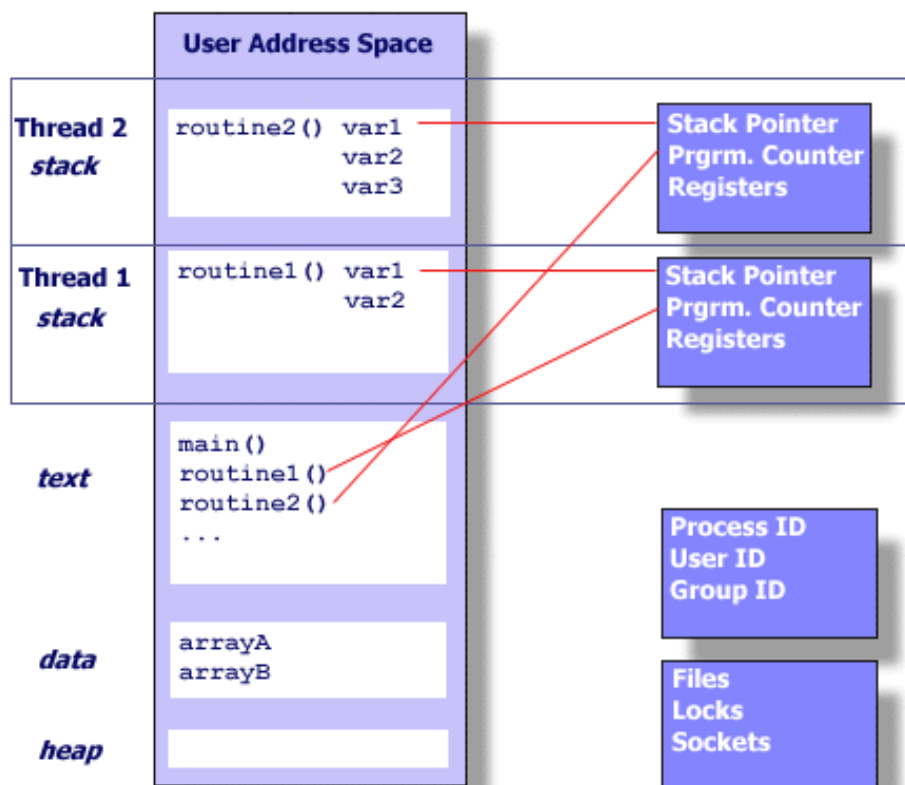
- Pilha (stack)
  - Mas atenção: não há isolamento entre pilhas!
  - *Bugs* podem fazer com que uma tarefa aceda à pilha de outra tarefa
- Estado dos registos do processador
  - Incluindo *instruction pointer*
- Atributos específicos da tarefa
  - Thread id (tid)
  - Etc (veremos mais tarde)

# Um processo Unix



em Posix Programming threads - <https://computing.lln.gov/tutorials/pthreads/>

# Um processo unix com duas tarefas (threads)



em Posix Programming threads - <https://computing.lnl.gov/tutorials/pthreads/>

# **Paralelismo com múltiplos processos vs.**

## **múltiplas tarefas (no mesmo processo)**

Quando faz sentido paralelizar um  
projeto com cada alternativa?

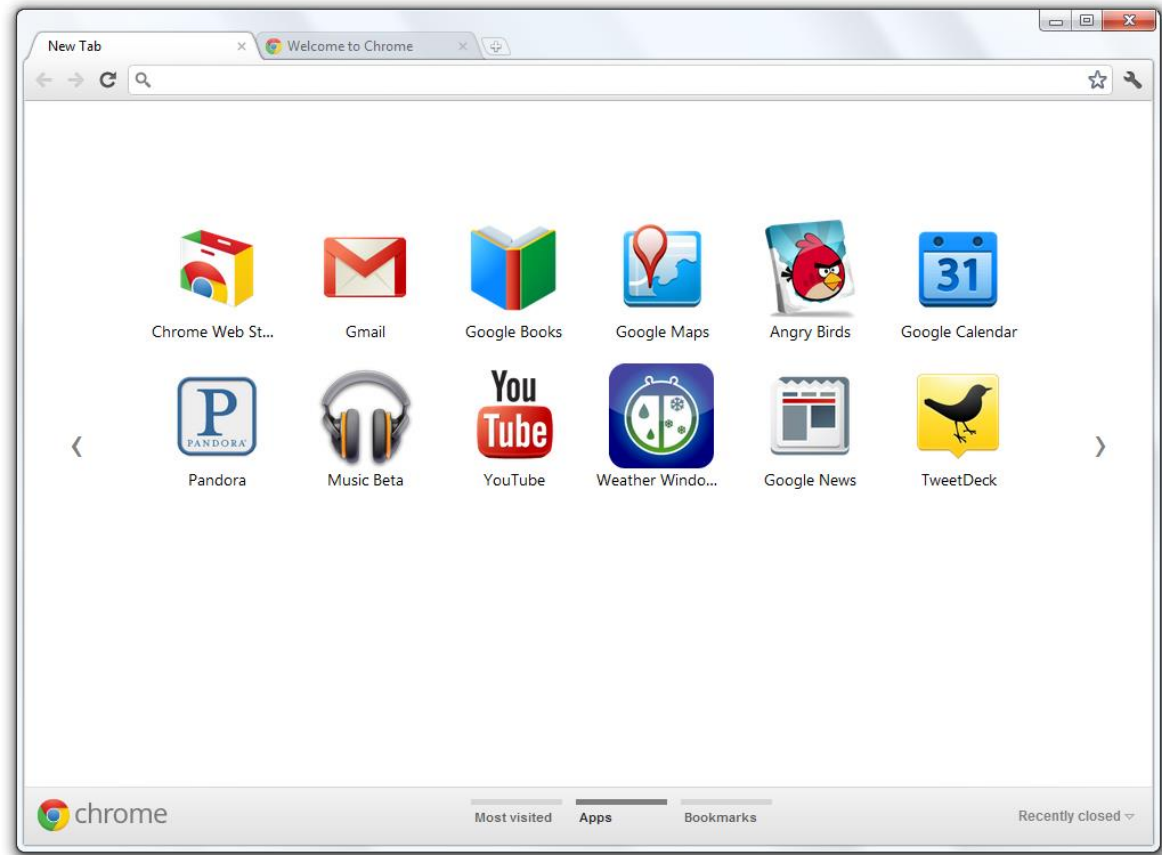


# Paralelizar com processos vs. tarefas (no mesmo processo)

- Vantagens de multi-tarefa:
  - Criação e comutação entre tarefas do mesmo processo mais leves (vs. entre processos)
  - Tarefas podem comunicar através de memória partilhada
    - Comunicação entre processos mais limitada, mas vamos aprender mais no futuro
- Vantagens de processos:
  - Podemos executar diferentes binários em paralelo
  - Isolamento: confinamento de *bugs*
  - Outras (veremos mais tarde)

# Exemplo de uso de processos: 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?





# Exemplo de uso de tarefas: PacmanIST!



# Programação de processos multi-tarefa em Unix/etc

Interface POSIX

# Interface POSIX: criar tarefa

`pthread_create(&tid, attr, function, arg)`

Apontador  
para o  
identificador  
da tarefa

Define atributos  
da tarefa  
(prioridade, etc)

Função a  
executar

Ponteiro  
para  
parâmetros  
para a  
função

# 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?

Esta regra também se aplica a programação com processos paralelos

# Exemplo: somar linhas de matriz

															$\Sigma$
															$\Sigma$
															$\Sigma$
															$\Sigma$
															$\Sigma$
															$\Sigma$
															$\Sigma$
															$\Sigma$



# Solução sequencial

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

int buffer [N] [TAMANHO];

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

    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

															$\Sigma$
															$\Sigma$
															$\Sigma$
															$\Sigma$
															$\Sigma$
															$\Sigma$
															$\Sigma$
															$\Sigma$



# Execução em N tarefas paralelas

															$\Sigma$
															$\Sigma$
															$\Sigma$
															$\Sigma$
															$\Sigma$
															$\Sigma$
															$\Sigma$
															$\Sigma$



# Exemplo (paralelo)

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

int buffer [N] [TAMANHO];

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

    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];

void *soma_linha (void *linha) {
    int c, soma=0;
    int *b = (int*) linha;

    for (c=0; c<TAMANHO-1; c++) {
        soma += b[c];
    }

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

```
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,
                           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);
}
```



# Criação de tarefa:

## Como passar/receber parâmetros?

- Parâmetros podem ser de **qualquer** tipo, passados por referência *opaca* (void\*)
- Parâmetro de entrada para a nova tarefa:
  - Através do argumento de *pthread\_create*
  - Nova tarefa recebe parâmetro no argumento único da sua função
- Parâmetro de saída devolvido pela nova tarefa
  - Função da tarefa retorna ponteiro para o parâmetro
  - Tarefa criadora recebe esse ponteiro através de *pthread\_join* (por referência)



# 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->userName);
    ...
}

int main (void) {

    MyStruct s;

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

        //Prepara argumentos da próxima thread
        s.userId = i;
        s.userName = getUserNam(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 ...
    }
    ...
}
```



# E neste programa?

```
void *threadFn(void *arg) {
    AnotherStruct r;
    ...
    r.x = ...;
    r.y = ...;
    return &r;
}

int main (void) {
    AnotherStruct *s2;

    //Cria thread, passando-lhe um user novo
    if (pthread_create (&tid, 0, threadFn, NULL) < 0) {
        ...
    }
    ...
    pthread_join(tid, &s2);
    printf("Tarefa devolveu: %d, %d\n", s2->x, s2->y);
}
```

**Então e se o programa não for  
embaraçosamente paralelo?**