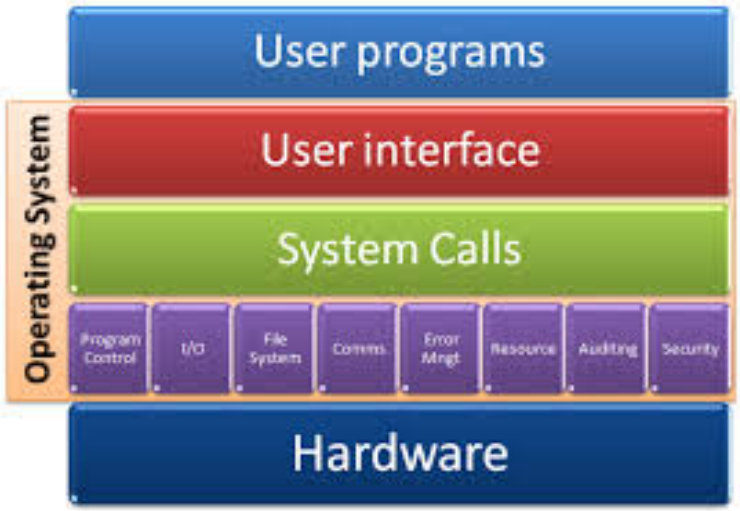
# Revisão dos Conceitos de Processos e *Threads*

João Marcelo Uchôa de Alencar
joao.marcelo@ufc.br
UFC-Quixadá

# Sistemas Operacionais




`https://www.bowdoin.edu/~sbarker/teaching/courses/os/18spring/`

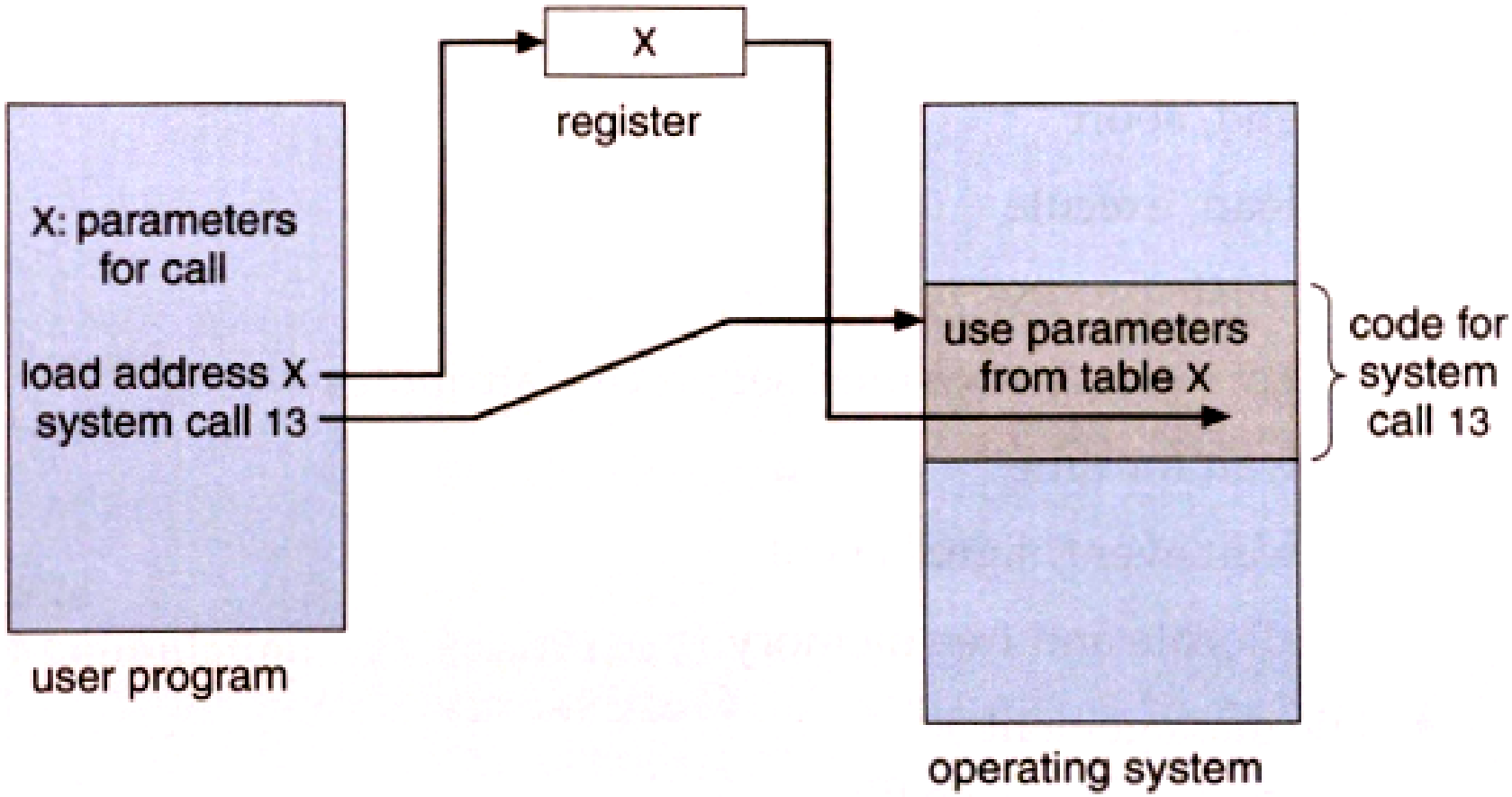
- Organizado em camadas;
- nível de usuário, *userspace*, *userland*, etc;
- nível de supervisor, *kernel*, *core*, etc;
- A comunicação entre o *userspace* e o *kernel* é estruturada em *chamadas de sistemas*.

# Chamadas de Sistemas

- Pense em chamadas de sistema como *funções* ou métodos que o *kernel* oferece;
- a maneira direta de acessar essas chamadas consiste em informar o número da chamada em um **registrador** e o endereço dos seus parâmetros em outro **registrador**;
- linguagens de alto nível fornecem bibliotecas que abstraem as chamadas de sistemas;
- nem sempre uma função na biblioteca equivale a uma **única** chamada de sistema.

Uma chamada de sistema envolve a **troca de contexto**, situação que consome tempo de execução.

# Chamadas de Sistemas




http://faculty.salina.k-state.edu/tim/ossg/Introduction/sys_calls.html

# Chamadas de Sistemas

- Execução de instruções não privilegiadas não invoca chamadas de sistemas.

```
/* Nenhuma troca de contexto. */
int a;
for (i = 0; i < 10000000; i++) {
    a = a + 1;
}
```

- qualquer ação de **sincronização**, **arquivos** e **rede** envolve chamadas de sistemas/troca de contexto;
- paginação/segmentação também podem acarretar troca de contexto.

A computação paralela tem como objetivo o desempenho.
Devemos tentar **minimizar** a troca de contexto.

# Exemplo de Chamadas de Sistemas

O exemplo a seguir mostra um programa simples em C:

- *common.h*;
- *cpu.c*;
- vamos compilá-lo no Linux e depois observar quais as chamadas de sistemas são feitas durante sua execução.

Mais na frente retornamos para discutir mais sobre compilação.

```c
/* common.h */
#ifndef __common_h__
#define __common_h__

#include <sys/time.h>
#include <sys/stat.h>
#include <assert.h>

double GetTime() {
    struct timeval t;
    int rc = gettimeofday(&t, NULL);
    assert(rc == 0);
    return (double) t.tv_sec + (double) t.tv_usec/1e6;
}

void Spin(int howlong) {
    double t = GetTime();
    while ((GetTime() - t) < (double) howlong)
        ; // do nothing in loop
}
#endif // __common_h__
```

```c
/* cpu.c */
#include <stdio.h>
#include <stdlib.h>
#include "common.h"

int main(int argc, char *argv[])
{
    if (argc != 2) {
            fprintf(stderr, "usage: cpu <string>\n");
            exit(1);
    }
    char *str = argv[1];

    while (1) {
            printf("%s\n", str);
            Spin(1);
    }
    return 0;
}
```

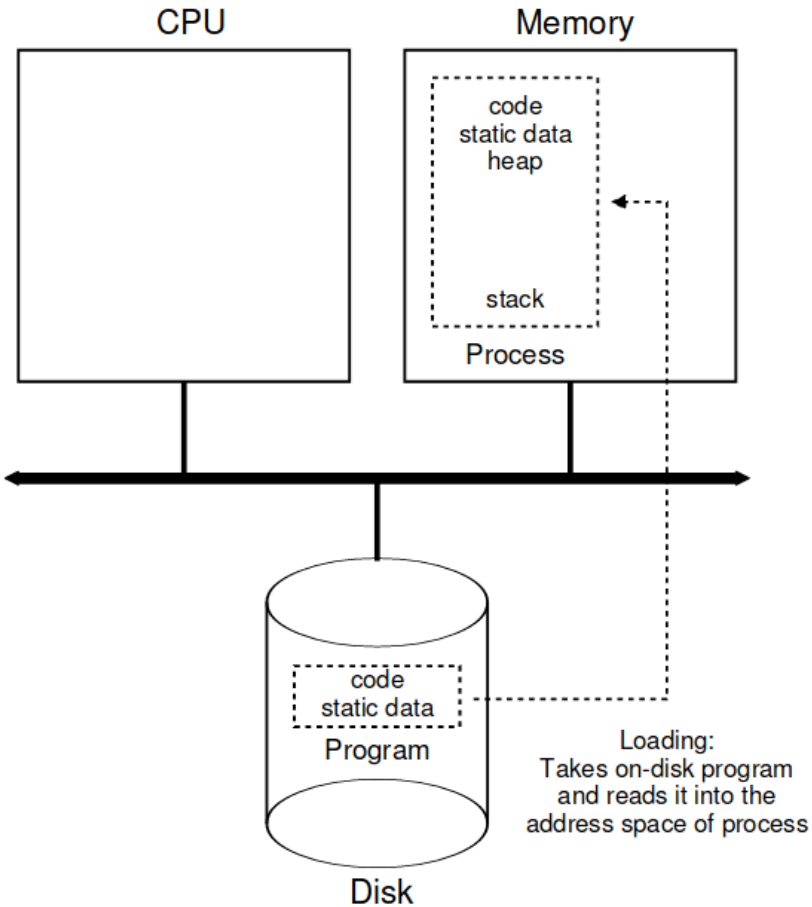
# Compilação do Exemplo

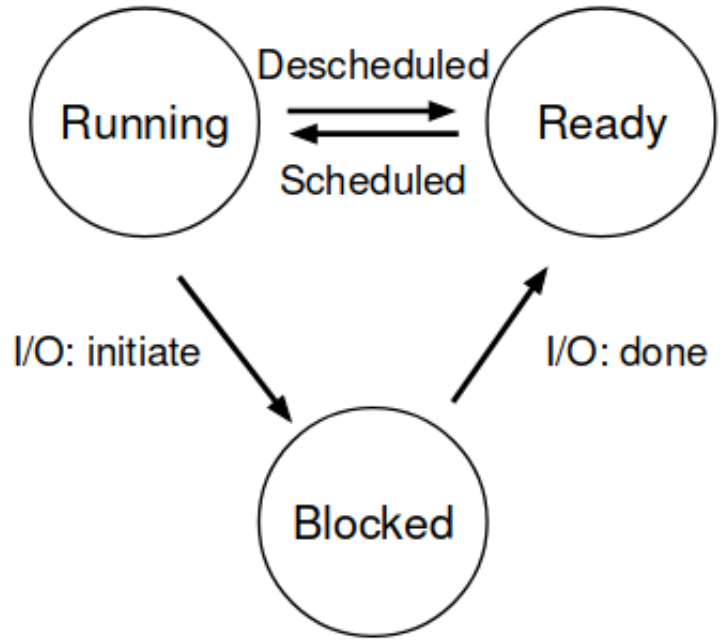
```
# Compilação e execução.
$ gcc -o cpu cpu.c
$ ./cpu teste
# Observando as chamadas de sistemas
$ strace ./cpu teste
```

https://github.com/remzi-arpacidusseau/ostep-code

# Processos

- Programa em execução;
- multiplexação da CPU por *virtualização*;
- contexto do processo:
  - Contador do programa;
  - espaço de endereçamento;
  - ponteiro para a pilha.
- o mesmo código (programa fonte) pode estar em execução em diversos processos diferentes.


CPU
Memory
code
static data
heap
stack
Process
code
static data
Program
Loading:
Takes on-disk program
and reads it into the
address space of process
Disk

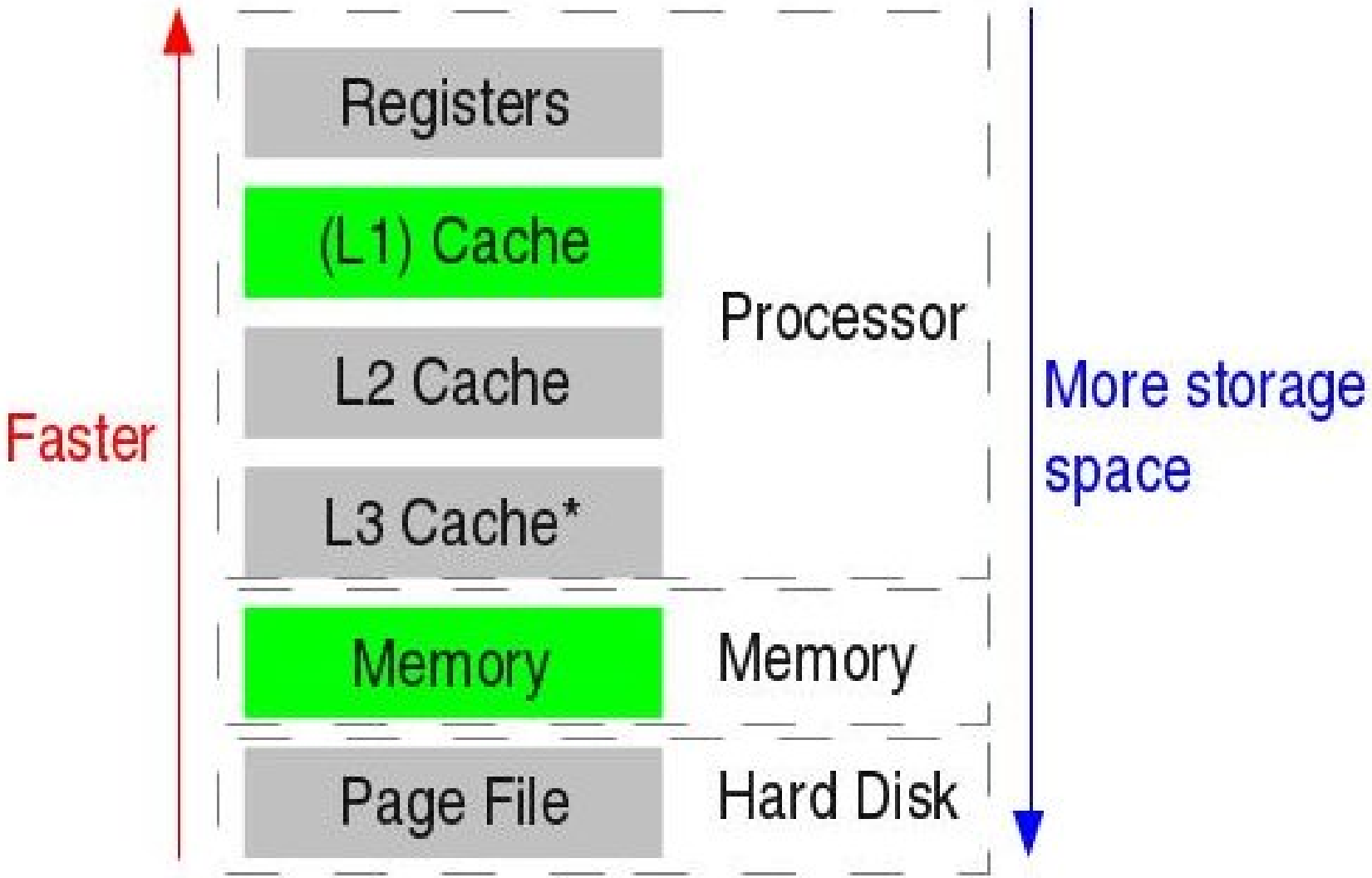

Descheduled
Running
Ready
Scheduled
I/O: initiate
I/O: done
Blocked

# Executando Vários Processo com Mesmo Código

```
$ ./cpu A & ./cpu B & ./cpu C & ./cpu D &
```

- Quatro processos são criados;
- cada um com seu contexto independente;
- como interromper a execução?

# Hierarquia de Memória



# Hierarquia de Memória

- A memória precisa ser dividida entre processos;
- os processos variam em tamanho durante sua execução;
- se faz necessário um sistema de tradução de endereços virtuais para endereços físicos.

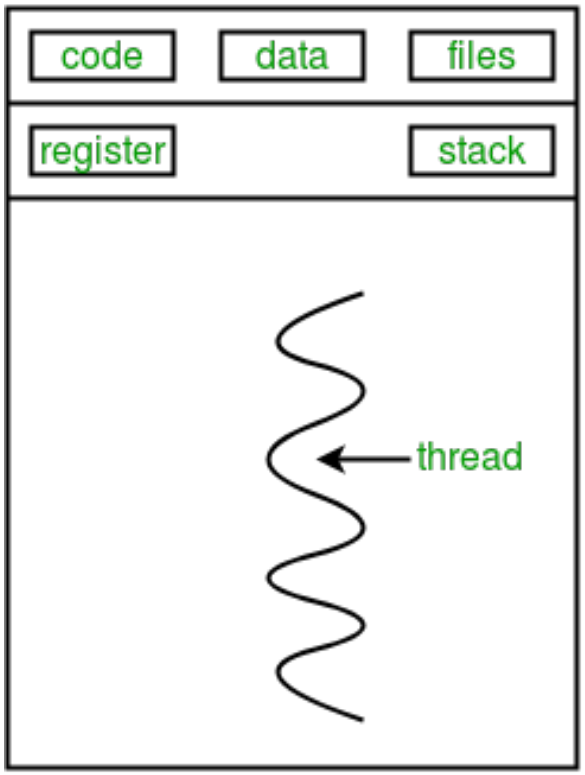
A paginação/segmentação (memória virtual) permitem programas maiores que a memória física.

- O programa pode ter partes na *cache*, memória principal, discos, etc;
- o ideal é que o processo sempre acesse o nível mais alto possível para manter um bom desempenho.

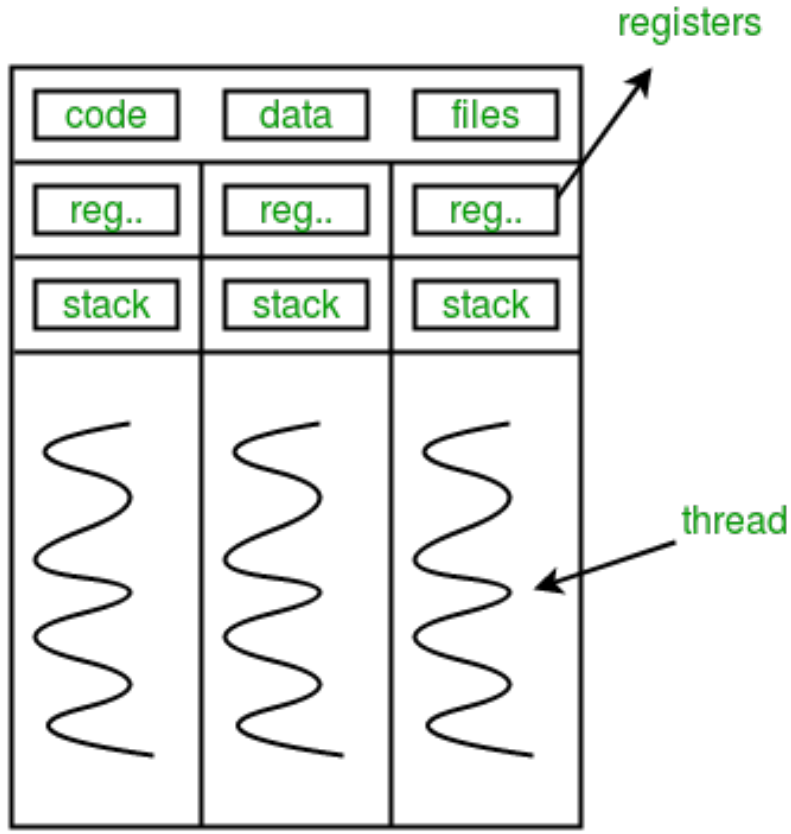
```c
/* mem.c */
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include "common.h"

int main(int argc, char *argv[]) {
    if (argc != 2) {
            fprintf(stderr, "usage: mem <value>\n");
            exit(1);
    }
    int *p;
    p = malloc(sizeof(int));
    assert(p != NULL);
    printf("(%d) addr pointed to by p: %p\n", (int) getpid(), p);
    *p = atoi(argv[1]); // assign value to addr stored in p
    while (1) {
            Spin(1);
            *p = *p + 1;
            printf("(%d) value of p: %d\n", getpid(), *p);
    }
    return 0;
}
```

# Threads




single-threaded process

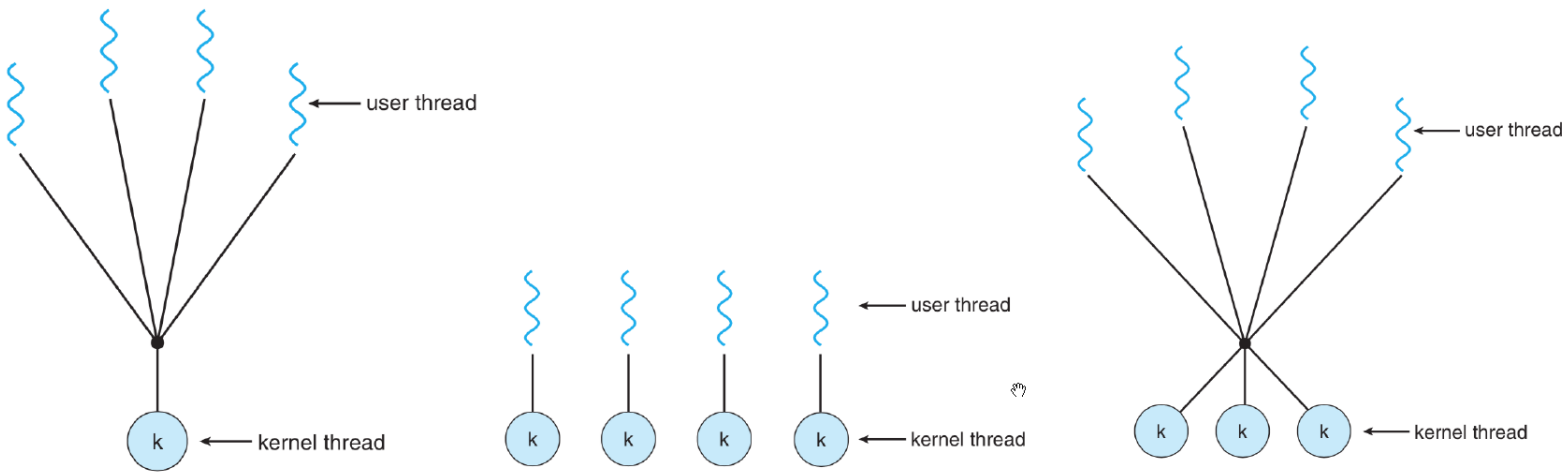



multithreaded process

`https://www.geeksforgeeks.org/multithreading-python-set-1/`

# Threads

- *Threads* permitem representar diferentes fluxos de execução em um mesmo processo;
- aliviam a sobrecarga da criação de processos;
- facilitam o compartilhamento de dados;




Left to right: User to Kernel level thread mappings: Many-To-One, One-To-One, and Many-To-Many [SGG13].

`https://csclub.uwaterloo.ca/~matedesc/figures/`

```c
/* threads.c */
#include <stdio.h>
#include <stdlib.h>
#include "common.h"
#include "common_threads.h"

volatile int counter = 0;
int loops;

void *worker(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
            counter++;
    }
    return NULL;
}

int main(int argc, char *argv[]) {
    if (argc != 2) {
            fprintf(stderr, "usage: threads <loops>\n");
            exit(1);
    }
    loops = atoi(argv[1]);
    pthread_t p1, p2;
    printf("Initial value : %d\n", counter);
    Pthread_create(&p1, NULL, worker, NULL);
    Pthread_create(&p2, NULL, worker, NULL);
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    printf("Final value   : %d\n", counter);
    return 0;
}
```

# Atividade

No seu repositório no *BitBucket*, crie uma pasta *atividades* e dentro dela uma pasta *atividade00* (deve ficar assim: *atividades/atividade00*). Coloque o código necessário para as tarefas abaixo:

- Os programas *cpu.c* e *mem.c* recebem como parâmetro um valor que é convertido para um inteiro.
- Altere os programas para receber mais um parâmetro, a ser convertido para inteiro e armazenado na variável *delay*.
- Essa variável deve ser fornecida como argumento para a a função *Spin* no lugar do valor fixo 1.