

# Нишки (threads)

проф. д-р инж. Христо Вълчанов

*<http://cs.tu-varna.bg>*

# Нишки

Нишка – независим поток от инструкции в рамките на една програма:

- Има различни състояния (running, ready и др.);
- Контекстът на нишката се съхранява когато тя не се изпълнява;
- Нишките се планират от ОС и се изпълняват като независими единици в рамките на процес;
- Нишките имат собствен стек;
- Един процес може да има множество нишки;
- Всяка нишка се представя чрез структура - thread control block (TCB);
- Всяка нишка може да създава други нишки.

# Процеси / нишки

- **Ключова разлика** - Множество нишки имат достъп до едно и също адресно пространство на процеса.

# Предимства на нишките

- **Най-важното предимство** - Достъп до адресното пространство на процеса.
- По-малко време за създаване.
- По-малко време за завършване.
- По-малко време за превключване между нишките.
- По-малко комуникационни разходи.

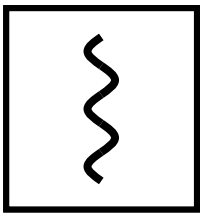
# Multi-threading / Single threading

- **Single threading**

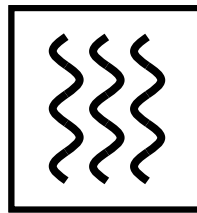
ОС не разпознава нишки.

- **Multi-threading**

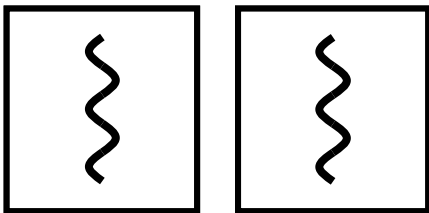
Поддържа нишки.



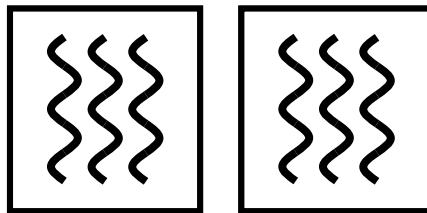
single process, single thread



single process, multiple threads



multiple processes, single thread

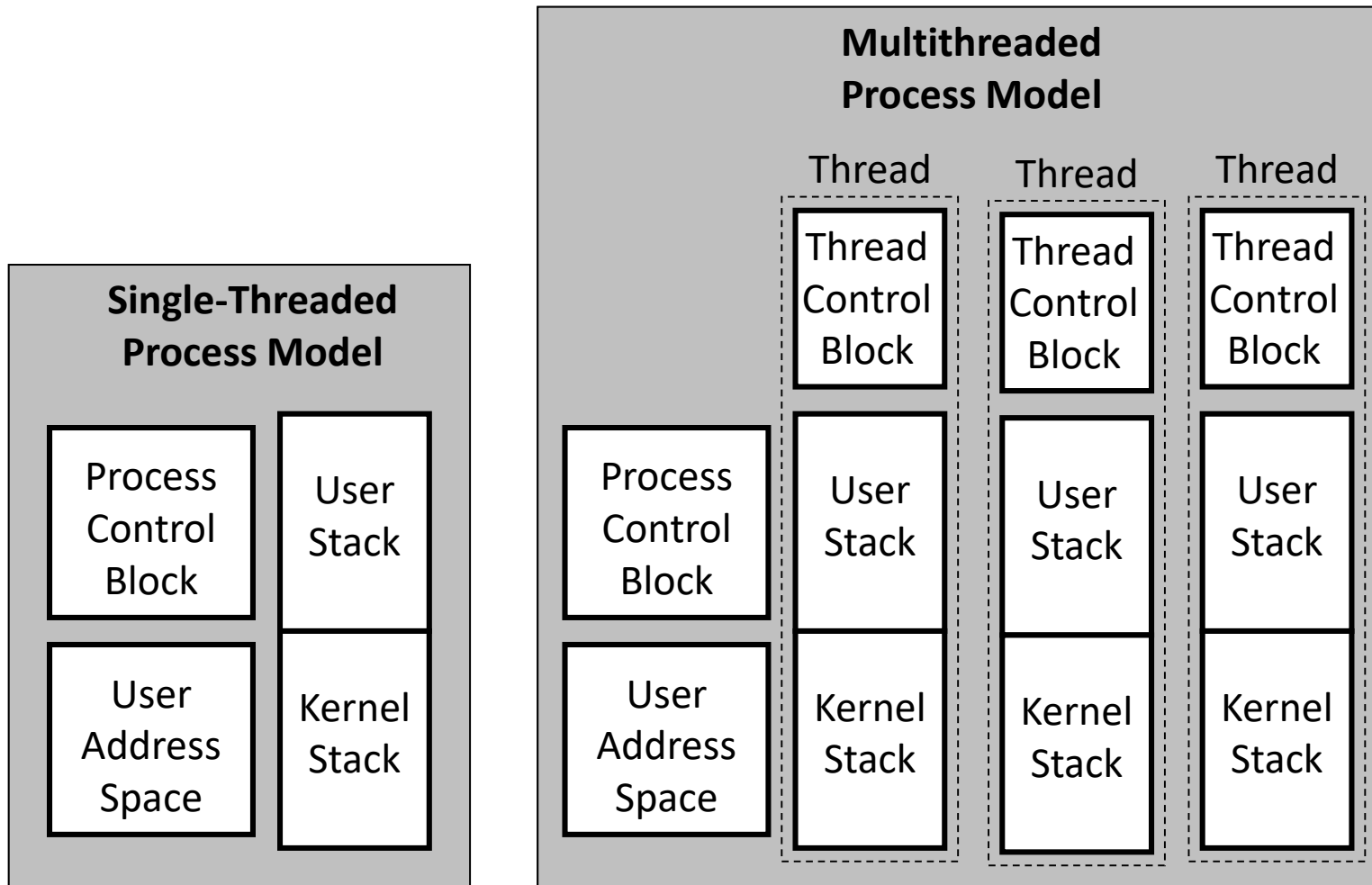


multiple processes, multiple threads

## Примери на ОС:

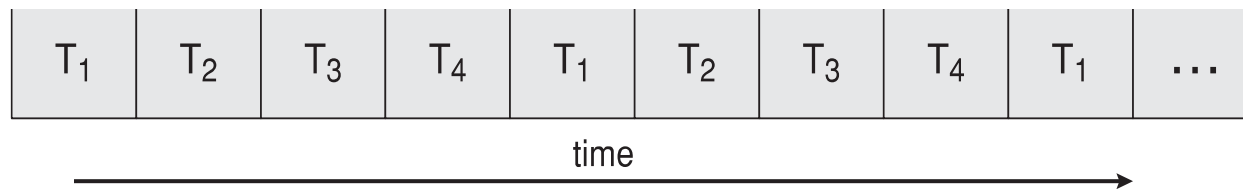
- **MSDOS** – един процес и една нишка
- **UNIX** – множество процеси с по една нишка всеки
- **Solaris, Windows** – множество нишки

# Модели на single и multithread приложения

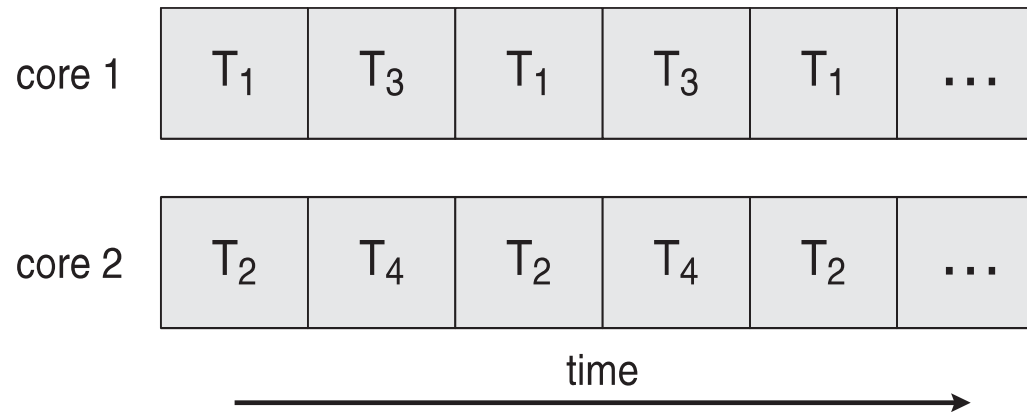


# Паралелно и конкурентно изпълнение

- Конкурентно изпълнение в едноядрена система



- Паралелно изпълнение в многоядрена система



# Multi-threaded клиент: Web Browsers





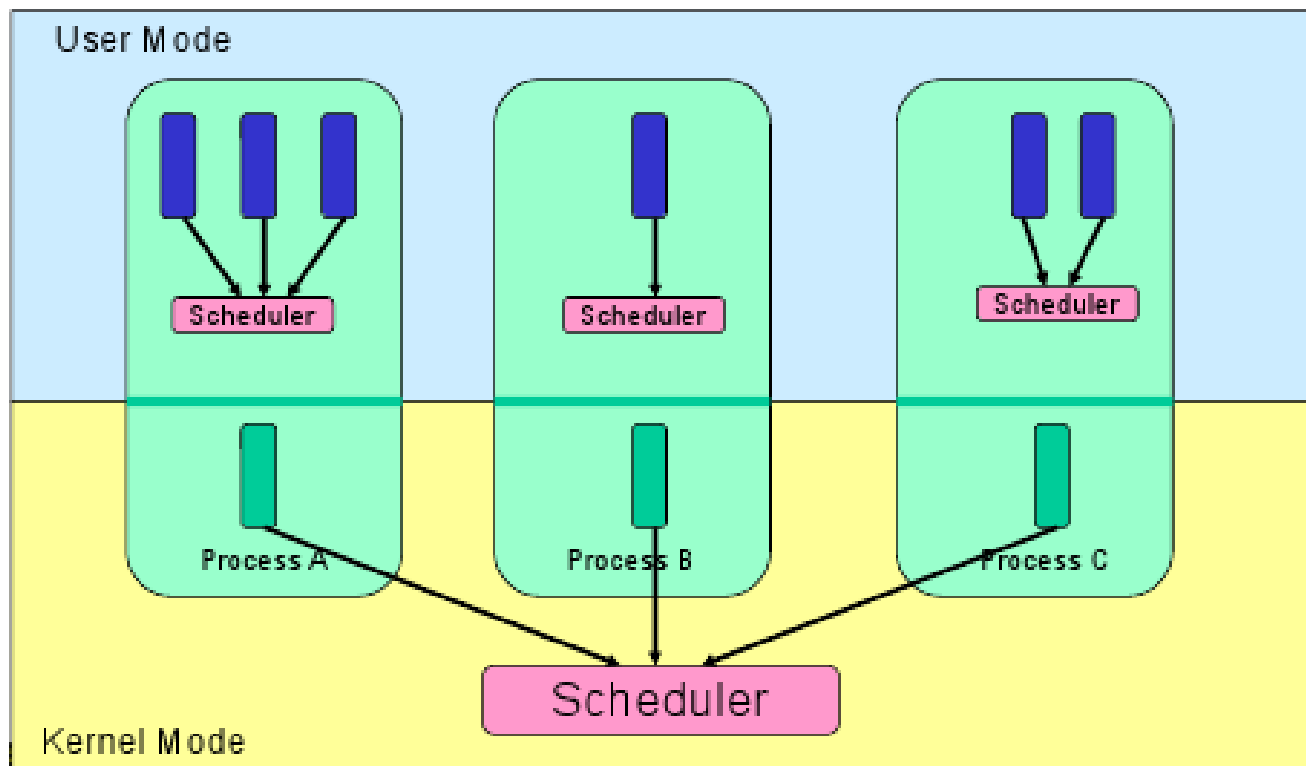
# Предимства на многонишковото програмиране

- Подобрява производителността на приложенията.
- Използване на многоядрени процесори.
- Подобрява структурата на програмите.
- Използват се по-малко системни ресурси.

# Реализация на нишки

- **User level** – ядрото не поддържа нишки. Управлението им е чрез библиотека.
- **Kernel level** – управлението на нишките се осигурява от ядрото.

# User-level нишки



# User-level – предимства и недостатъци

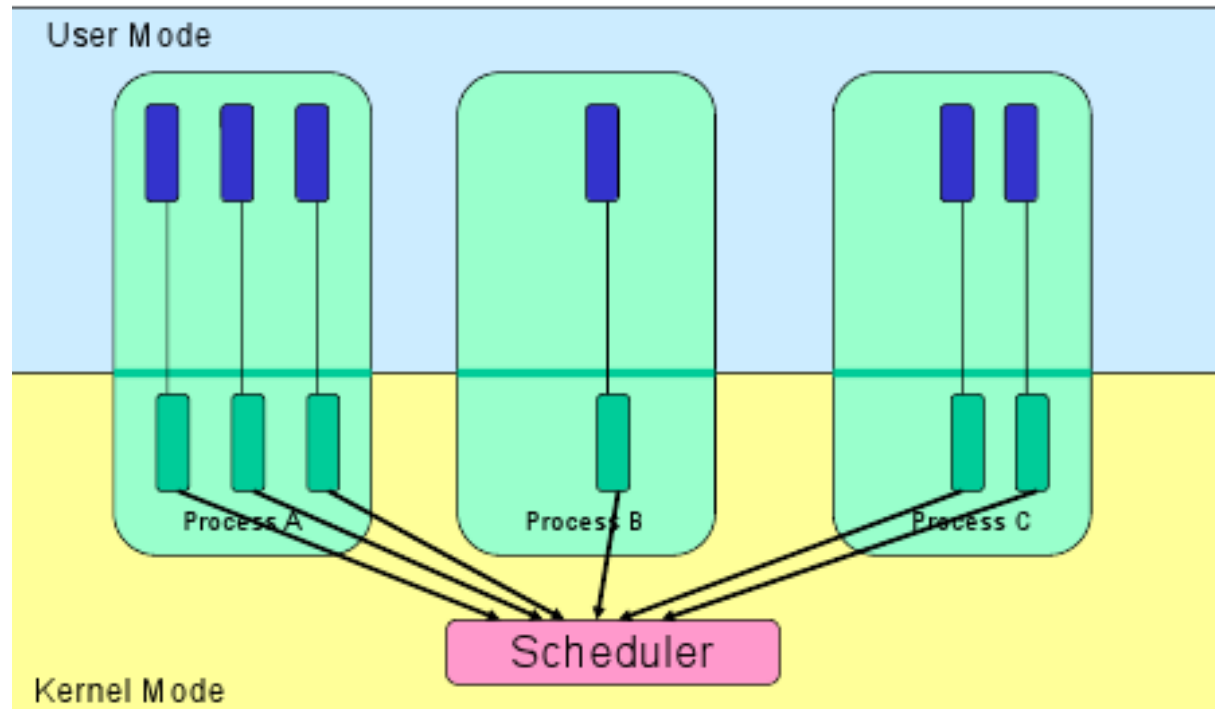
## Предимства:

- Ядрото не участва в превключванията на нишките.
- Планирането може да е приложно ориентирано – избор на най-добрия алгоритъм.
- Нишките могат да се изпълняват на всяка ОС – необходима е само библиотека.

## Недостатъци:

- Повечето системни извиквания за блокиращи и ядрото блокира процеса – всички нишки в него се блокират.
- Ядрото може да присъединява само процеси към процесори. Нишките в процес не могат да се изпълнят паралелно на няколко процесора.

# Kernel-level нишки



# Kernel-level – предимства и недостатъци

## Предимства:

- Ядрото може да планира едновременно много нишки. от същия процес върху няколко процесора.
- Блокирането се извършва на ниво нишка.
- Функциите на ядрото могат да бъдат реализирани многонишково.

## Недостатъци:

- Превключването в рамките на един процес се извършва от ядрото. Резултатът е забавяне на изпълнението.

# Библиотеки за нишки

- Предоставят интерфейс за работа с НИШКИ.
  - **libpthread** for POSIX threads
  - **libthread** for Solaris threads
  - Java Threads
  - Win 32 Threads

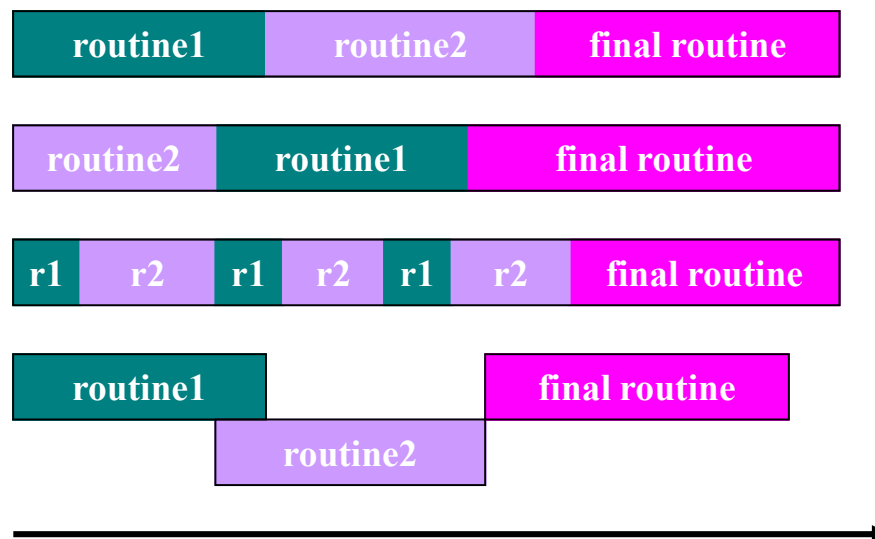
# Пакети за нишки

- Posix Threads (pthreads)
  - Широко използван
  - Базиран на Posix standard
  - Опростени извиквания: *pthread\_create*, ...
  - Типично използван в C/C++ приложения
  - Може да бъде реализиран и като user-level и като kernel-level
- Java Threads
  - Естествена поддръжка, вградена в езика
  - Нишките се планират от JVM
- Win 32 Threads



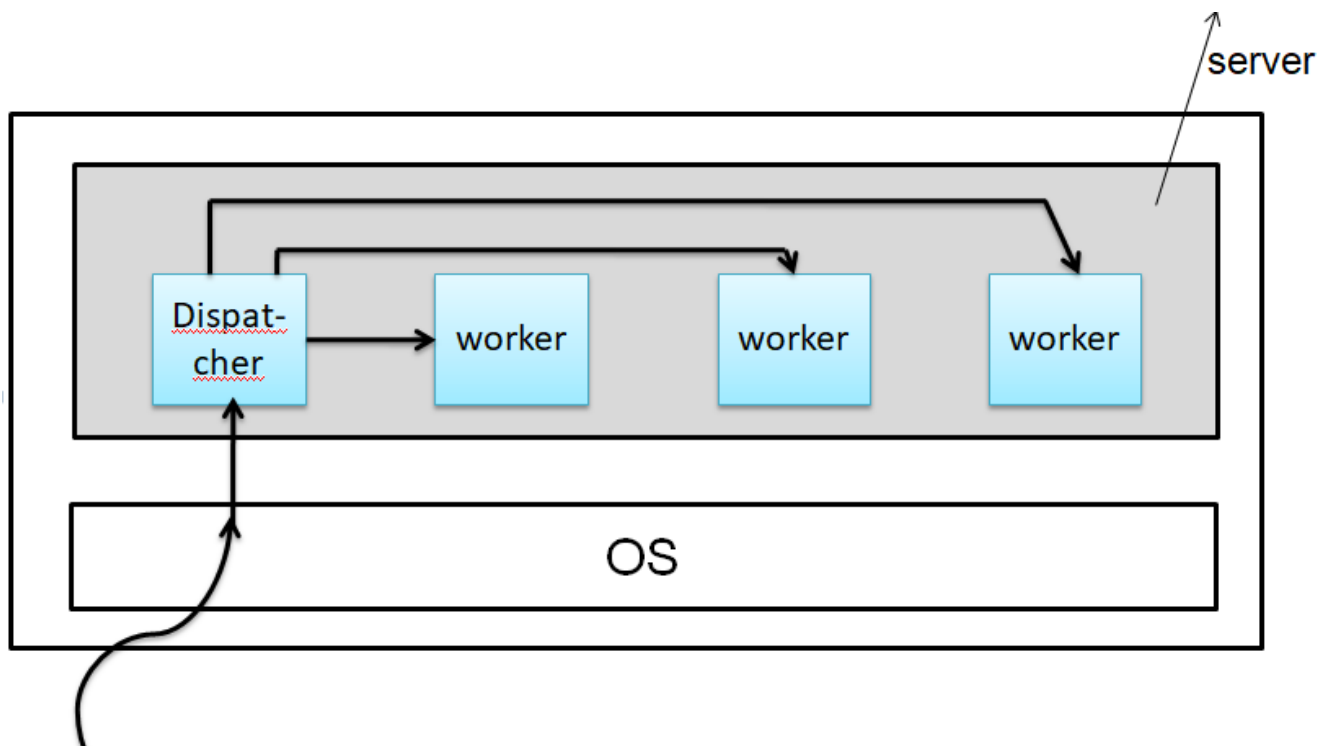
# Проектиране на многонишкови програми

- Ако програмата може да се организира в независими, конкурентно изпълняващите се единици.



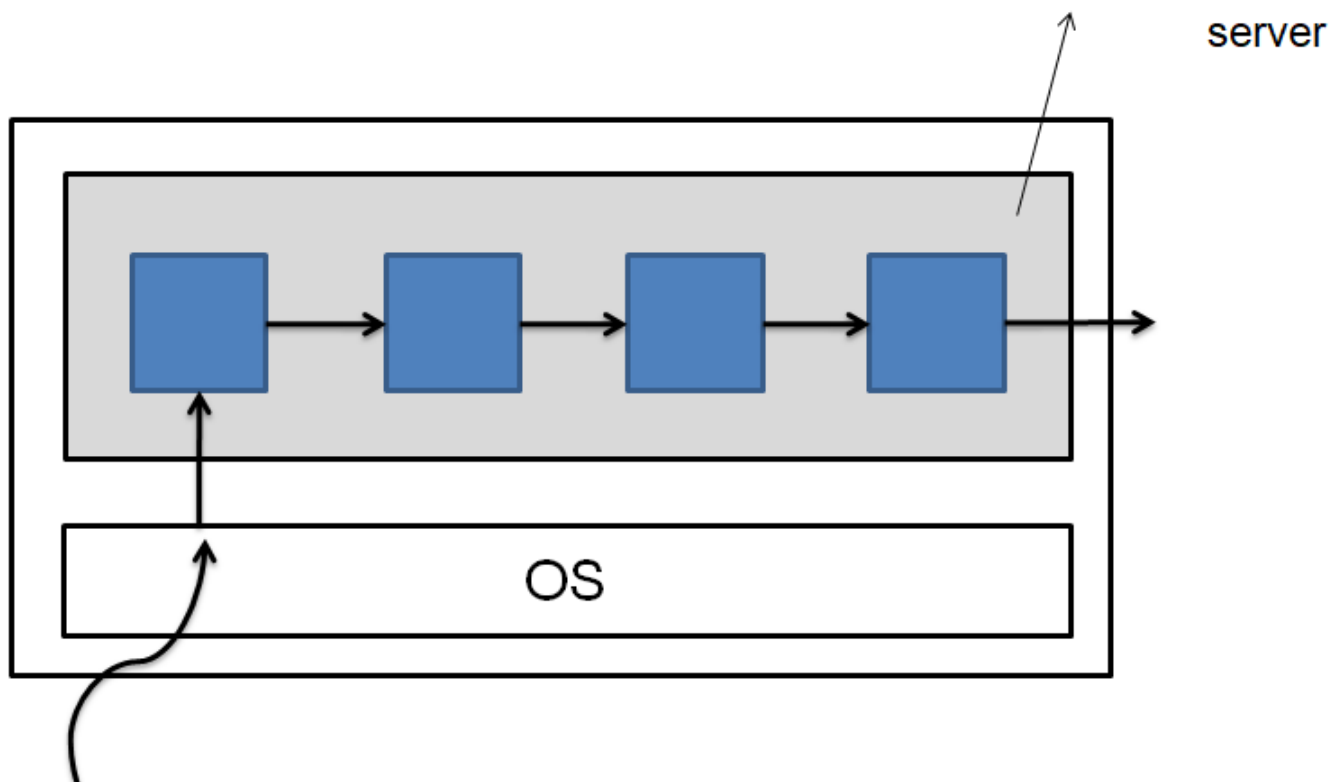
# Общи модели на многонишкови програми

- **Manager-worker**



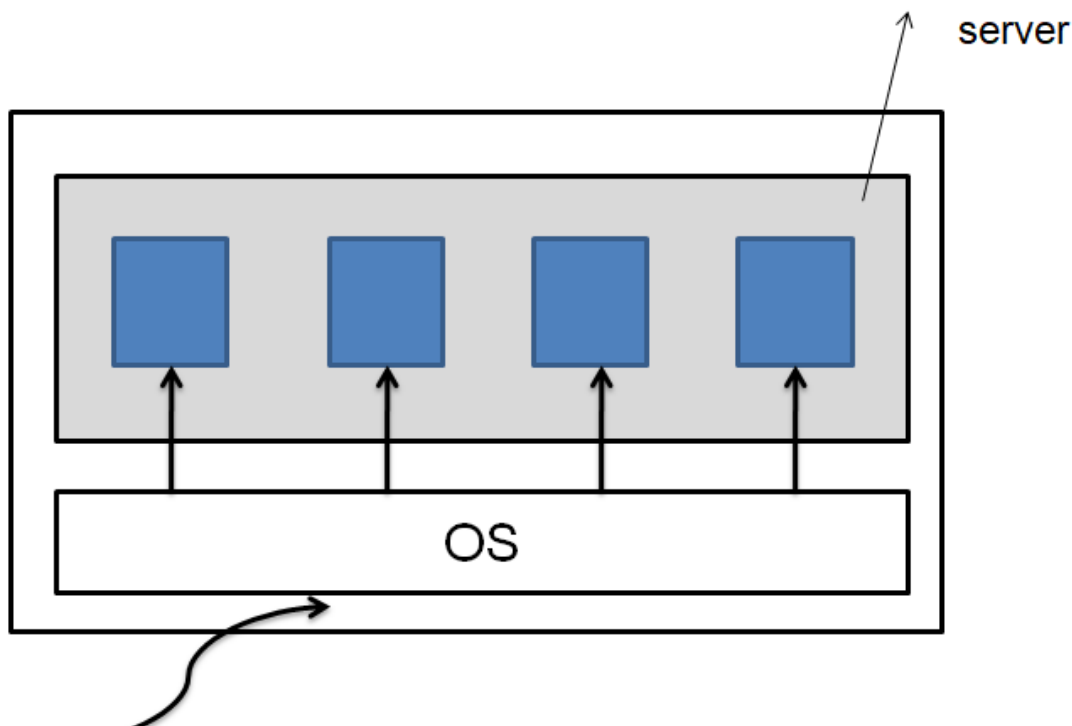
# Общи модели на многонишкови програми

- **Pipeline**



# Общи модели на многонишкови програми

- Team server



# Pthreads API

- **Управление на нишки**
- **Mutexes**
- **Conditional variables**

# Компилиране на многонишкова програма

```
> gcc -o <executable_name> <source_file> -lpthread
```

Необходимо е задаване на библиотеката *pthread* с опцията -l

# Създаване на нишка

```
int pthread_create(pthread_t *tid,  
                  const pthread_attr_t *tattr,  
                  void* (*start_routine)(void *),  
                  void *arg);
```

# Завършване на нишка

- При завършване на функцията, от която е създадена;
- При извикване на *pthread\_exit()*;
- Процесът завършва чрез *exit()*.

```
void pthread_exit(void *status) ;
```



# Пример

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS      5

void *PrintHello(void *threadid){
    printf("\n%d: Hello World!\n", threadid);
    pthread_exit(NULL);
}

int main (int argc, char* argv[]){
    pthread_t threads[NUM_THREADS];
    int rc, t;

    for(t=0; t < NUM_THREADS; t++){
        printf("Creating thread %d\n", t);

        // Create a thread and pass its number as argument
        rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);

        if (rc){
            printf("ERROR: pthread_create %d\n", rc);
            exit(1);
        }
    }

    pthread_exit(NULL);
}
```

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

## Пример 2

```
void *Plus(void *threadid){
    int k;
    for ( ; ; ) {
        printf("+");
        for (k=0;k<10000;k++);
    }
}

void *Minus(void *threadid){
    int k;
    for ( ; ; ) {
        printf("-");
        for (k=0;k<10000;k++);
    }
}

void *Dot(void *threadid){
    int k;
    for ( ; ; ) {
        printf(".");
        for (k=0;k<10000;k++);
    }
}
```

```
int main (int argc, char* argv[]){
    pthread_t t1, t2, t3;

    printf("Creating thread %d\n", t);

    pthread_create(&t1, NULL, Plus, NULL);
    pthread_create(&t2, NULL, Minus, NULL);
    pthread_create(&t3, NULL, Dot, NULL);

    pthread_exit(NULL);
}
```

## Пример 2 - проблем

```
int main (int argc, char* argv[]){  
    pthread_t t1, t2, t3;  
  
    printf("Creating thread %d\n", t);  
  
    pthread_create(&t1, NULL, Plus, NULL);  
    pthread_create(&t2, NULL, Minus, NULL);  
    pthread_create(&t3, NULL, Dot, NULL);  
  
    return (0);  
}
```

## Пример 2 - проблем

```
int main (int argc, char* argv[]){  
    pthread_t t1, t2, t3;  
  
    printf("Creating thread %d\n", t);  
  
    pthread_create(&t1, NULL, Plus, NULL);  
    pthread_create(&t2, NULL, Minus, NULL);  
    pthread_create(&t3, NULL, Dot, NULL);  
  
}
```

# Изчакване на нишка

```
int pthread_join(thread_t tid,  
                  void **status);
```

# Пример

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS      3

// A working thread
void *BusyWork(void *null) {
    int i;
    double result=0.0;

    // Simulate working
    for (i=0; i < 1000000; i++) {
        result = result + (double)random();
    }
    printf("result = %e\n",result);
    pthread_exit(NULL);
}
```

# Пример

```
int main (int argc, char* argv[]) {
    pthread_t thread[NUM_THREADS];
    int rc, t, status;

    for(t=0; t < NUM_THREADS; t++) {
        printf("Creating thread %d\n", t);

        // Create a thread as JOINABLE
        rc = pthread_create(&thread[t],
                           NULL, BusyWork, NULL);

        if (rc) {
            printf("ERROR: pthread_create %d\n", rc);
            exit(1);
        }
    }
}
```

# Пример

```
for(t=0;t < NUM_THREADS;t++) {  
    // Wait for other treads  
    rc = pthread_join(thread[t], (void *)&status);  
    if (rc) {  
        printf("ERROR: pthread_join %d\n", rc);  
        exit(1);  
    }  
    printf("Main: Completed join with thread %d  
           status= %d\n",t,status);  
}  
  
pthread_exit(NULL);  
}
```



# Синхронизация на нишка

- Mutual Exclusion locks (Mutex).
- Semaphores.
- Condition variables.

# Mutexes - операции

- **lock()**
- **unlock()**
- **trylock()**

# Използване

`lock mutex;`

*Critical Section;*

`unlock mutex;`

# Инициализиране и унищожаване на mutex

```
int pthread_mutex_init(pthread_mutex_t *mp,  
                        const pthread_mutexattr_t *mattr);
```

✓ **mutex е първоначално отключен**

```
int pthread_mutex_destroy(pthread_mutex_t *mp);
```

# Заклучване/отключване на mutex

```
int pthread_mutex_lock(pthread_mutex_t *mp);
```

```
int pthread_mutex_unlock(pthread_mutex_t *mp);
```

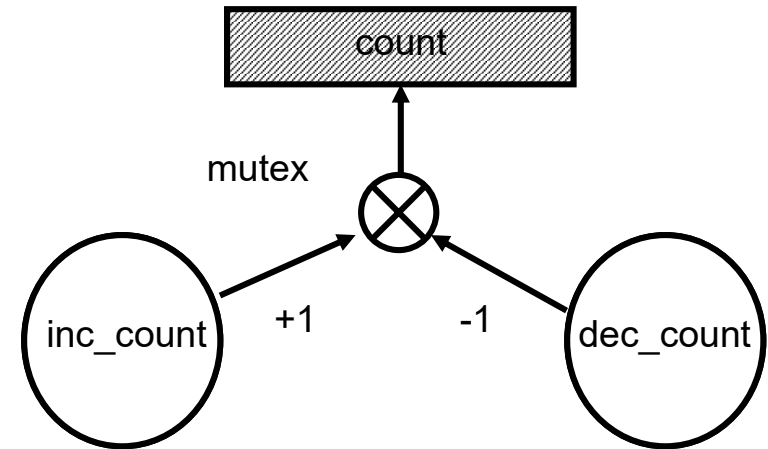
# Пример

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

int count=0;
pthread_mutex_t mu;    // Mutex variable

void *inc_count(void* arg) {
    int i=0;
    for (i=0;i<1000000;i++) {
        pthread_mutex_lock(&mu);
        count++;
        pthread_mutex_unlock(&mu);
        if (i%100000==0) printf("+\n");
    }
    pthread_exit(NULL);
}

void *dec_count(void* arg) {
    int i=0;
    for (i=0;i<1000000;i++) {
        pthread_mutex_lock(&mu);
        count--;
        pthread_mutex_unlock(&mu);
        if (i%100000==0) printf("-\n");
    }
    pthread_exit(NULL);
}
```



```
int main(int argc, char* argv[]) {
    pthread_t t1, t2;

    // Initialize the mutex
    pthread_mutex_init(&mu, NULL);

    pthread_create(&t1, NULL, inc_count, NULL);
    pthread_create(&t2, NULL, dec_count, NULL);

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    // Free the mutex resources
    pthread_mutex_destroy(&mu);

    printf("Main: Done. Count = %d\n", count);

    pthread_exit(NULL);
}
```

# Семафори - операции

```
int sem_init(sem_t *sem,  
             int pshared,  
             unsigned int value);
```

```
int sem_wait(sem_t *sem);
```

```
int sem_post(sem_t *sem);
```

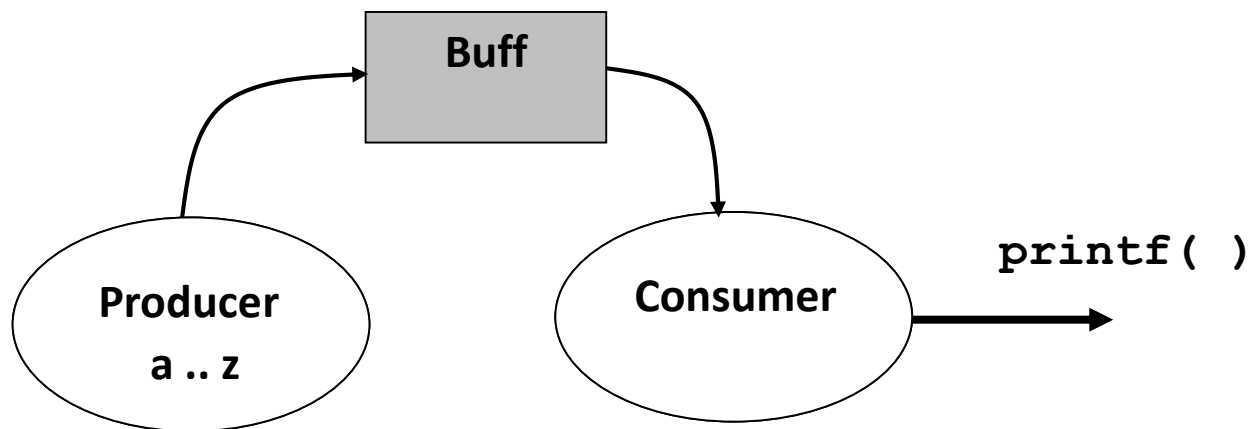
```
int sem_getvalue(sem_t *sem, int *sval);
```



# Пример “Производител-консуматор”

Производителят генерира азбуката и я записва символ по символ в буфер от тип *char*.

Консуматорът чете данните от буфера и ги извежда на екрана.



```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <pthread.h>
#include <semaphore.h>
#include <signal.h>

char buffer = ' '; // The buffer

void *Producer (void *arg) {
    int k, x, i;

    for ( i='a'; i<='z'; i++ ){
        x = 0;
        for ( k=0; k<100000000; k++ ) x++;
        buffer = i;    // Critical section
    }
    printf("Producer thread ended.\n");
    pthread_exit(NULL);
}
```

```
void *Consumer (void *arg) {
    int k, x, i;
    char a;

    for ( i='a'; i<='z'; i++ ) {

        a = buffer;    // Critical section

        printf("%c\n", a);
        if ( a == 'z' ) break;
        x = 0;
        for ( k=0; k<5000000; k++ ) x++;
    }
    printf("Consumer thread ended.\n");
    pthread_exit(NULL);
}
```

```
int main(int argc, char *argv[]) {
    pthread_t proc, cons;
    int status;

    // Create Producer and Consumer threads
    status = pthread_create(&proc, NULL, Producer, NULL);
    if (status != 0) {
        perror("Create thread Producer");
        exit(1);
    }
    status = pthread_create(&cons, NULL, Consumer, NULL);
    if (status != 0) {
        perror("Create thread Consumer");
        exit(1);
    }

    printf("Main thread ended. \n");
    pthread_exit(NULL);
}
```

# Резултат

Възможни варианти на извеждането:

- а а а а а а а b b c d d d d d d d e e ...
- b f p z
- z

# Решение с двоичен семафор

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <pthread.h>
#include <semaphore.h>
#include <signal.h>

char buffer = ' ';

sem_t mutex;

void *Producer (void *arg) {
    int k, x, i;

    for ( i='a'; i<='z'; i++ ){
        x = 0;
        for ( k=0; k<10000000; k++ ) x++;
        sem_wait( &mutex );
        buffer = i;      // Critical section
        sem_post( &mutex );
    }
    printf("Producer thread ended.\n");
    pthread_exit(NULL);
}

void *Consumer (void *arg) {
    int k, x, i;
    char a;

    for ( i='a'; i<='z'; i++ ) {
        sem_wait( &mutex );
        a = buffer;      // Critical section
        sem_post( &mutex );
        printf("%c\n", a);
        if ( a == 'z' ) break;
        x = 0;
        for ( k=0; k<5000000; k++ ) x++;
    }
    printf("Consumer thread ended.\n");
    pthread_exit(NULL);
}
```

```

int main(int argc, char *argv[]) {
    pthread_t proc, cons;
    int status;

    if (sem_init(&mutex, 0, 1) == -1) {
        perror("Init semaphore");
        exit(1);
    }

    // Create Producer and Consumer threads
    status = pthread_create(&proc, NULL, Producer, NULL);
    if (status != 0) {
        perror("Create thread Producer");
        exit(1);
    }

    status = pthread_create(&cons, NULL, Consumer, NULL);
    if (status != 0) {
        perror("Create thread Consumer");
        exit(1);
    }

    // Wait for threads to complete
    pthread_join(proc, NULL);
    pthread_join(cons, NULL);

    printf("Main thread ended.\n");
    pthread_exit(NULL);
}

```

# Решение със семафори

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <pthread.h>
#include <semaphore.h>
#include <signal.h>

char buffer = ' ';

sem_t mutex;
sem_t full;
sem_t empty;

void *Producer (void *arg) {
    int k, x, i;

    for ( i='a'; i<='z'; i++ ){
        x = 0;
        for ( k=0; k<10000000; k++ ) x++;
        sem_wait( &empty );
        sem_wait( &mutex );
        buffer = i;          // Critical section
        sem_post( &mutex );
        sem_post( &full );
    }
    printf("Producer thread ended.\n");
    pthread_exit(NULL);
}

void *Consumer (void *arg) {
    int k, x, i;
    char a;

    for ( i='a'; i<='z'; i++ ) {
        sem_wait( &full );
        sem_wait( &mutex );
        a = buffer; // Critical section
        sem_post( &mutex );
        sem_post( &empty );
        printf("%c\n", a);
        if ( a == 'z' ) break;
        x = 0;
        for ( k=0; k<5000000; k++ ) x++;
    }
    printf("Consumer thread ended.\n");
    pthread_exit(NULL);
}
```

---



```
int main(int argc, char *argv[]) {
    pthread_t proc, cons;
    int status;

    if (sem_init(&mutex, 0, 1) == -1) {
        perror("Init semaphore");
        exit(1);
    }

    if (sem_init(&empty, 0, 1) == -1) {
        perror("Init semaphore");
        exit(1);
    }

    if (sem_init(&full, 0, 0) == -1) {
        perror("Init semaphore");
        exit(1);
    }

    . . .

}
```

# Многонишков сървър

```
#include <stdio.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <pthread.h>
#include <stdlib.h>

#define SVRPORT          3000

// Worker thread function
void *worker(void *arg) {
    int D1, D2, operation, sum;
    int sd_copy;

    // The argument is the socket copy from accept()
    sd_copy = *(int*)arg;

    // Read from the socket
    read(sd_copy, &D1, sizeof(D1));
    read(sd_copy, &D2, sizeof(D2));
    read(sd_copy, &operation,
        sizeof(operation));

    switch (operation) {
        case 0: sum = D1 + D2; break;
        case 1: sum = D1 - D2; break;
        case 2: sum = D1 * D2; break;
        case 3: sum = D1 / D2; break;
        case 4: exit(0);
        default: {
            printf("Server: Error\n");
            exit(1);
        }
    }

    write(sd_copy, &sum, sizeof(sum));

    // Return the result

    close(sd_copy);

    pthread_exit(0); // End of the worker
}
```

# Многонишков сървър

```
int main() {
    int sd, sd_copy;
    struct sockaddr_in myaddr, theclient;
    int clientlen, tmp=1;
    pthread_t td;

    // Create a socket
    sd = socket(AF_INET, SOCK_STREAM, 0);
    if (sd == -1) {
        perror("socket");
        exit(1);
    }

    // Bind to the socket
    myaddr.sin_family = AF_INET;
    myaddr.sin_port = htons(SVRPORT);
    myaddr.sin_addr.s_addr = htonl(INADDR_ANY);

    if ( bind(sd, (struct sockaddr *)&myaddr,
        sizeof(myaddr)) == -1 ) {
        perror("bind");
        exit(1);
    }

    // Set socket to be with REUSABLE port
    if ( setsockopt(sd, SOL_SOCKET, SO_REUSEADDR,
        &tmp, sizeof(int)) == -1 ) {
        perror("setsockopt");
        exit(1);
    }

    // Prepare buffer for incoming connections
    if ( listen(sd, 5) == -1 ) {
        perror("listen");
        exit(1);
    }

    // Server's main loop
    while (1) {
        // Wait for a client connection
        sd_copy = accept(sd,
            (struct
            sockaddr*)&theclient,
            &clientlen);
        if (sd_copy == -1) {
            perror("accept");
            exit(1);
        }

        // Start worker thread
        pthread_create(&td, NULL, worker,
            (void *)&sd_copy);
    }

    exit(0);
}
```

# Въпроси?