

The background of the entire slide is a top-down view of numerous colorful donuts scattered across a light blue surface. The donuts are in various colors including yellow, orange, red, purple, teal, green, and pink. Some are plain, while others are topped with sprinkles or chocolate chips. A dark purple, arrow-shaped banner points from the left towards the center, containing the title and authors' names.

Σειρά εργασιών 4

ΣΤΑΜΟΥΛΟΣ ΑΛΕΞΑΝΔΡΟΣ

ΚΥΡΙΤΣΗΣ ΒΑΣΙΛΕΙΟΣ


ΑΣΚΗΣΗ 1

```
int mycoroutines_init(co_t *main){
    getcontext(&Main.ctx);
    return 0;
} // initialize main's context

int mycoroutines_create(co_t *co, void(body)(void*) ,void *arg){
    getcontext(&co->ctx);
    co->ctx.uc_link = &Main.ctx; // the coroutine terminates it return to main
    co->ctx.uc_stack.ss_sp = malloc(MEM); // allocate memory for coroutine's stack
    co->ctx.uc_stack.ss_size = MEM;
    //create
    makecontext(&co->ctx, (void*) body, 1, arg); //όταν κανουμε swap context η εκτελεση του
    προγραμματος θα συνεχιστει καλώντας την body(arg)
    return 0;
}

int mycoroutines_switchto(co_t *start ,co_t *co){
    swapcontext(&start->ctx, &co->ctx); //swap context from start to co
    return 0;
}

void mycoroutines_destroy(co_t* co){
    free(co->ctx.uc_stack.ss_sp); // deallocate the stack memory
}
```



```
Void coroutine1() {
    while(1) {
        if(buffer is full)
            switch(&co1,&co2);
        scan a char from file
        if scan == EOF
            break;
        put char in ring buffer
    }
    switch(&co1,&co2);
}
```

```
Void coroutine2() {
    While(1) {
        if(buffer is empty)
            switch(&co2,&co1);

        get char from buffer
    }
}
```



ΑΣΚΗΣΗ 2

Χρησιμοποιούμε ένα implementation ενός queue με το ακόλουθο API:

```
void queue_init (queue *que);  
int enqueue(queue *que, thr_t *thread);  
thr_t *dequeue(queue *que);  
int que_size (queue *que);  
thr_t *get_q_ele (queue *que, int num); //get i-th element  
int remove_q_ele(queue *que, thr_t *thread);
```

Όστε να αποθηκεύουμε τα threads που βρίσκονται στο ready queue και στο finish queue.

Έχουμε ένα clock-alarm που στέλνει το signal SIGALRM κάθε 10μs



```
void my_thread_init() {  
    queue_init(ready_q);  
    queue_init(finish_q);  
    set signal handler for SIGALRM  
  
    start_time();  
    getcontext from main  
  
    current = main; // thr_t *current -> current running thread  
  
}
```



Scheduler for threads

Enter when we receive SIGALRM

```
Void scheduler() {  
    stop_time();  
    prev = current;  
    next_running_thread = dequeue from ready queue  
    current = next_running_thread  
    start_time();  
    swapcontext(prev, current);  
  
}
```

Tuple Space

```
typedef struct t_node{  
    int type; TYPE : int 0, char 1, string 2, scan_int 3, scan char 4, scan_string 5  
    int d;  
    char c;  
    char s[20];  
    struct t_node *next;  
}t_node;
```

To κάθε tuple είναι μια linked list από t_node.

To tuple space είναι μια linked list από tuples.

Προσθήκη /Αφαίρεση πλειάδας στο tuple space

Παίρνει παρόμοια ορίσματα με την printf();

Πχ: Για την προσθήκη της πλειάδας <'α', 4, "paiktes"> χρησιμοποιούμε
`tuple_out("cds", 'a', 4, "paiktes");`

Αντίστοιχα για το tuple in, παίρνει ορίσματα παρόμοια με την scanf();


Πχ: Για την αφαίρεση της πλειάδας <'α', 4, "paiktes" > χρησιμοποιούμε
`tuple_in("cds", 'a', 4, "paiktes");`

Ενώ όταν θέλουμε να αποθηκεύσουμε σε μια μεταβλητή κάποια τιμή από το tuple space χρησιμοποιούμε στα ορίσματα το % πχ:

`tuple_in("c%d%s", 'a', &d, s);`

Εδώ δηλαδή θα γίνουν d = 4 , s = "paiktes".

Οι συναρτήσεις tuple_out, tuple_in είναι thread safe γιατί κάνουν stop time, δηλαδή δεσμεύουν τον επεξεργαστή μέχρι την προσθήκη / αφαίρεση πλειάδας.



```
void tuple_in(char *fmt, ...) {  
    stop_time();  
    while(can't find tuple in tuple space) {  
        start_time();  
        yield();  
        stop_time();  
    }  
    start_time();  
    return;  
}
```