ECS 150 - Project 2

Prof. Joël Porquet-Lupine

UC Davis - WQ22

Organization

Assignment

- Assignment released on Wednesday
- Due in ~two weeks: Feb 10th
- Two parts
    - Queue API
    - User-level thread library

Teamwork

Two options:

1. Keep the same partner as P1
    - *(But you'll have to change partners for P3)*
2. Change partner
    - *(And you'll be able to keep the same new partner for P3)*

Goal #1: queue API

- Most pieces of software need typical data structure implementations
    - Lists, stacks, queues, hashtables, dictionaries, etc.

- In many languages, directly included in language itself
    - Python (`mylist = ["apple", "banana", "cherry"]`), Javascript, Perl, etc.

- In some other languages, usually need to develop them from scratch
    - Especially true in C and in system programming
        - Linux: `linux/include/linux/list.h`
        - Glibc: `glibc/include/list.h`
        - GCC: `gcc/gcc/lists.c`
    - Need very robust containers:
        - Clear API,
        - No memory leaks,
        - Stable,
        - Flexible,
        - Etc.

For this project, implement a queue library which API is provided

Pointers

`void *`, a special kind of pointer

```c
int queue_enqueue(queue_t queue, void *data);
int queue_dequeue(queue_t queue, void **data);

void myfunc(queue_t q)
{
    /* Various objects */
    short int a = 2;
    char *b = "P2 will teach me threads";
    struct {
        int stuff;
        char thing;
    } c = { 10, 'c' };
    int *d = malloc(10 * sizeof(int));

    /* Push in queue */
    queue_enqueue(q, &a);
    queue_enqueue(q, b);
    queue_enqueue(q, &c);
    queue_enqueue(q, d);

    /* Retrieve from queue */
    short int *e;
    char *f;
    queue_dequeue(q, (void**)&e);
    queue_dequeue(q, (void**)&f);
    ...
}
```

- `void*` is an **untyped** pointer
- Contains an address, but can't assume the type of the pointed content

Pointers

Double pointers

- A double pointer is simply the address of a pointer.

```c
int a = 2;
int b = 4;

void change_ptr(int **ptr)
{
    *ptr = &b;
}

void change_val(int *ptr)
{
    *ptr += 1;
}

int main(void)
{
    int *c = &a;

    printf("%d\n", *c);

    change_val(c);
    printf("%d\n", *c);

    change_ptr(&c);
    printf("%d\n", *c);

    return 0;
}
```

Execution

```
2
3
4
```

Pointers

Variables vs pointers (1)

```c
void swap(int x, int y)
{
    int tmp = x;
    x = y;
    y = tmp;
}

int a = 2, b = 4;

printf("%d, %d\n", a, b);

swap(a, b);

printf("%d, %d\n", a, b);
```

```
2, 4
2, 4
```

```c
void swap(int *x, int *y)
{
    int tmp = *x;
    *x = *y;
    *y = tmp;
}

int a = 2, b = 4;

printf("%d, %d\n", a, b);

swap(&a, &b);

printf("%d, %d\n", a, b);
```

```
2, 4
4, 2
```

Pointers

Variables vs pointers (2)

```
void swap(void *x, void *y)
{
    void *tmp = x;
    x = y;
    y = tmp;
}

int a = 2, b = 4;
int *c = &a, *d = &b;

printf("%d, %d\n", *c, *d);

swap(c, d);

printf("%d, %d\n", *c, *d);
```

```
2, 4
2, 4
```

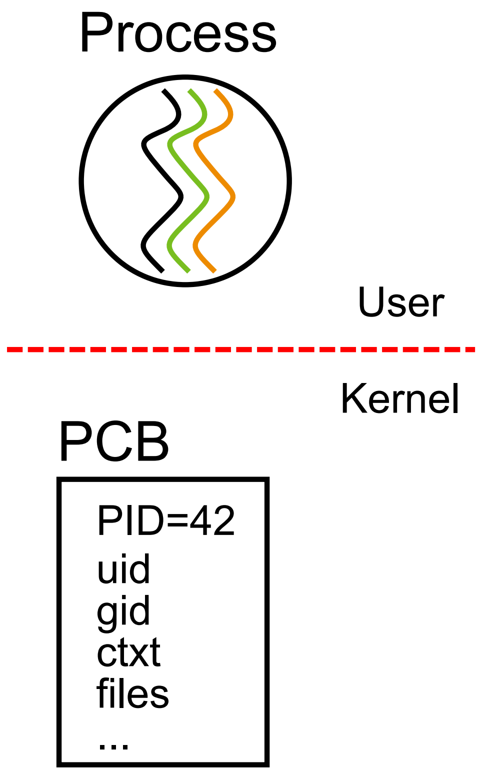
```
void swap(void **x, void **y)
{
    void *tmp = *x;
    *x = *y;
    *y = tmp;
}

int a = 2, b = 4;
int *c = &a, *d = &b;

printf("%d, %d\n", *c, *d);

swap(&c, &d);

printf("%d, %d\n", *c, *d);
```

```
2, 4
4, 2
```

Goal #2: uthread library

Write a thread library at user level

- Kernel sees and schedules one process
- Inside the process, can schedule multiple threads of execution




```c
#include <uthread.h>

int thread_b(void) {
    /* Do stuff... */
    return 0;
}

int thread_a(void) {
    uthread_t tid = uthread_create(thread_b);
    /* Do stuff... */
    uthread_join(tid, NULL);
    return 0;
}

int main(void) {
    uthread_start(0);
    uthread_join(uthread_create(thread_a), NULL);
    uthread_stop();

    return 0;
}
```

ECS 150 - Project 2

Prof. Joël Porquet-Lupine

UC Davis - WQ22

Queue implementation

Demystify `void *`

Step 1

If struggling with `void *` for your implementation

- Copy `queue.{c,h}` elsewhere
- Temporarily replace all `void *` with `int`
- Implement a queue of integer with the suggested API

```c
int queue_enqueue(queue_t queue, int data);
int queue_dequeue(queue_t queue, int *data);
int queue_delete(queue_t queue, int data);
...
```

Step 2

- Once your implementation is solid
- Replace `int` by `void *`
  - `:%s/int/void */g`
- Copy back into project 2

API vs implementation

Uthread library

- In directory `libuthread`
- API in headers
  - `private.h` *(internal to the library)*
  - `queue.h`, `uthread.h`
- Implementation of API in C file
  - `context.c`, `preempt.c`, `queue.c`, `uthread.c`
- Local Makefile generates a static library that contains the compiled code
  - `libuthread.a`
  - At first, add only `queue.o`
  - Then, add the other objects

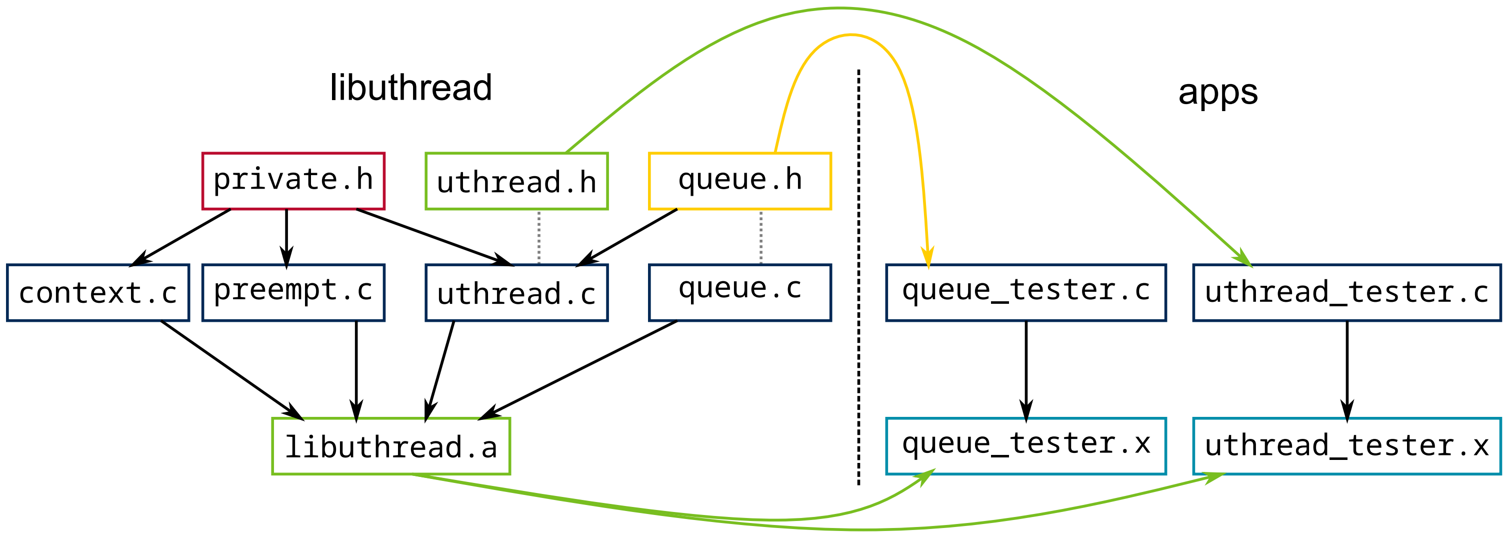
API vs implementation

Applications

- In directory `apps`
- Applications written by "external" users using the API
- Access to library through its public API
  - `#include <queue.h>`
  - `#include <uthread.h>`
- No access to the library's internals
  - Otherwise implementation can never evolve without breaking compatibility with existing applications
  - Goal is that the applications should work with *any* implementation
  - **NEVER** include C files in the applications
- Makefile combines the library with the applications' code
  - `-L/path/to/libuthread -luthread`

API vs implementation

Illustrated summary



Testing the queue

Test the API

Correctness

- Verify that the implementation actually respects the API
  - E.g., enqueue item, then dequeue it and verify it's the same
- Build various scenarios, from very simple to complex

Error management

- Explore *all* the possible "mistakes" a user could do and check the implementation behaves according to the specs
  - E.g., try to enqueue a `NULL` item, see if function returns proper error code

Overall

- Should come up with 10 to 20 *different* unit tests
  - Ideally get to 100% coverage
- Look at `/home/cs150jp/public/p2/queue_tester_example.c` to get an idea

Testing the queue

Debug

Printing

1. Add static helper function in queue.c (eg, `queue_print()`), call it from the other functions.
2. **Temporarily** add helper function in public API, call it from the applications. Remove entirely when done debugging.

GDB

- Breakpoints, step-by-step execution
    - `$ make D=1`

    ```
    ## Debug flag
    ifneq ($(D),1)
    CFLAGS    += -O2
    else
    CFLAGS    += -g
    endif
    ```

join() mechanism

Simple example

```c
int thread1(void)
{
    printf("thread1\n");
    return 5;
}

int main(void)
{
    int ret;
    uthread_t tid;

    uthread_start(0);

    tid = uthread_create(thread1);
    uthread_join(tid, &ret);
    printf("thread1 returned %d\n", ret);

    uthread_stop();

    return 0;
}
```

```
$ ./a.out
thread1
thread1 returned 5
```

1. main runs until join
    - Creates thread1 which is added to the ready queue
    - Gets blocked in join
2. thread1 runs entirely
    - Next available thread in ready queue
    - Context switch from main
    - Exits with return value 5
3. main() runs until the end
    - When thread1 died, unblocked main
    - Retrieved return value
    - Main is added back to the ready queue and scheduled
    - Resume previous execution

join() mechanism

Complex example

```c
uthread_t tid[2];

int thread2(void)
{
    int ret;
    printf("thread2\n");
    uthread_join(tid[0], &ret);
    printf("thread1 returned %d\n", ret);
    return 2;
}

int thread1(void)
{
    tid[1] = uthread_create(thread2);
    printf("thread1\n");
    return 1;
}

int main(void)
{
    int ret;

    uthread_start(0);
    tid[0] = uthread_create(thread1);
    uthread_yield();
    uthread_join(tid[1], &ret);
    printf("thread2 returned %d\n", ret);
    uthread_stop();

    return 0;
}
```

- main cannot be joined
- For other threads, the parent/child relationship is not relevant

- The uthread library doesn't care about the return values, just provides the transmission mechanism