HDU-ITMO Joint Institute
杭州电子科技大学 圣光机联合学院

Distributed Computing
Laboratory work #1
Introduction to communication framework

Michael Kosyakov
Associate Professor

Denis Tarakanov
Assistant Lecturer

dstarakanov@itmo.ru
ifmo.distributedclass.bot@gmail.com

Classes plan

1. Model of distributed computing
2. Communication framework API
3. Task for laboratory work 1
4. Bot usage
5. Control questions

Model of distributed system

- What is distributed system?
- Important moments:
  - Lack of common memory
  - Communication via network by using messages
- Multiple processes system as model of distributed system
  - Address spaces of processes are separated
  - Processes can communicate via IPC mechanism by using messages of predefined protocol

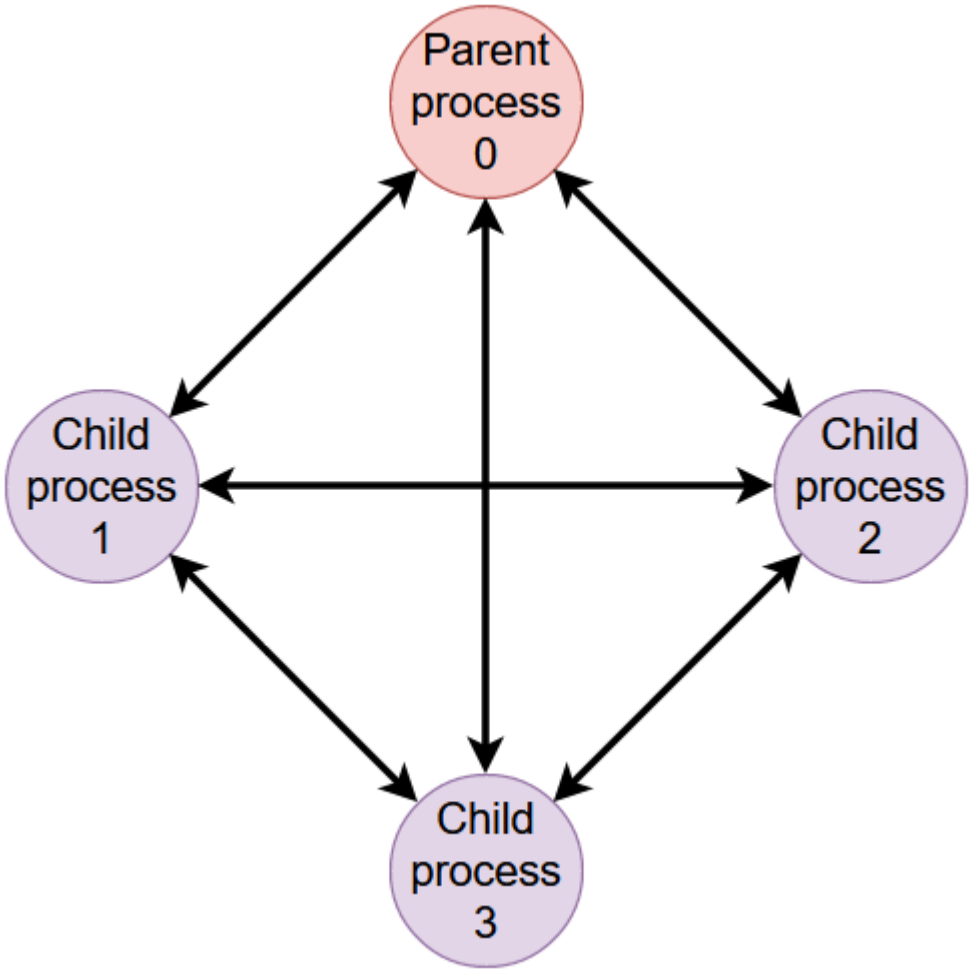
Model of distributed system

- libdistributedmodel.so
  - Framework, implemented model of distributed system
  - Initialize required processes
  - Provides communication primitives for interaction between processes

- System structure
  - Fully connected topology
  - Processes can have different responsibilities
  - One parent process, leader of the group, participate in synchronizations
  - N child processes, which are responsible for “useful” work
  - All communications between processes are based on FIFO principle

Model of distributed system

- Example for 3 child processes



Communication framework API

- Presented in header files from labs_headers
- Documentation for functions and structures is available as commentaries in header files
- Functions and structures from file banking.h aren't used in this laboratory work

- process.h
  - Provides signatures of functions, which should be implemented by students
- message.h
  - Provides API for sending messages between processes and structures, which define communication protocol
- log.h
  - Provides functions for logging and formats of required logs

Header file "process.h"

- `void parent_work(int count_nodes);`
  - Represents the main execution function of parent process
  - Should be implemented by students
  - Arguments:
    - count_nodes - total count of all processes (child and parent)
      - E.g from previous example it will be 4
  - Note, self_id of parent process defined by constant PARENT_ID and always equal to 0

Header file “process.h”

- `void child_work(struct child_arguments args);`
  - Represents the main execution function of any child process
  - Should be implemented by students
  - Arguments:
    - args - arguments of this child process
      - See description of structure on the next slide

Header file "process.h"

```
struct child_arguments {
    local_id self_id;
    int count_nodes;
    uint8_t balance;
    bool mutex_usage;
};
```

- Defines required information about child process
- Fields:
  - self_id - internal id of current process (unique for this process)
  - count_nodes – total count of processes (child and parent)

  For 1 parent and 3 children count_nodes will be equal to 4
  - balance – unused in this laboratory work, equal to 0
  - mutex_usage – unused in this laboratory work, equal to false

Header file "message.h"

- `int send(local_id dst, const Message * msg);`
- `int receive_any(Message * msg);`
  - This functions aren't used for the first laboratory work and will be introduced later

Header file "message.h"

- `int send_multicast(const Message * msg);`
  - Send message to all other processes including parent
  - Arguments:
    - msg – pointer to message, which should be sent
      - See description of structure on the next slide

Header file "message.h"

- `int receive(local_id from, Message * msg);`
  - Receive a message from the process specified by id
  - Call to this function will block execution until message from process with id "from" is received
  - If process with id "from" is already terminated – framework will stop execution of program with related error message
  - Arguments:
    - from - id of the process to receive message from
    - msg – pointer to message structure, allocated by user, where received message should be stored
      - See description of structure on the next slide

Header file "message.h"

```
typedef struct {
    MessageHeader s_header;
    char s_payload[MAX_PAYLOAD_LEN];
} __attribute__((packed)) Message;
```

- Defines a format of message in the distributed system
- Fields:
  - s_header – structure, represented header of a message, which holds technical information
    - See description of structure on the next slide
  - s_payload – buffer for useful content of a message
    - Example of usage will be provided on slide 20

Header file "message.h"

```
typedef struct {
    uint16_t      s_magic;
    uint16_t      s_payload_len;
    int16_t       s_type;
    timestamp_t   s_local_time;
} __attribute__((packed)) MessageHeader;
```

- Defines a format of message header
- Fields:
  - s_magic – magic signature, must be MESSAGE_MAGIC
  - s_payload_len – length of payload, which is sent
  - s_type – type of the message
  - s_local_time – time of message sending event, unused in this laboratory work, should be 0

Header file "message.h"

```
void fill_message(Message * msg,
                  MessageType type,
                  timestamp_t time,
                  void * payload,
                  size_t psize);
```

- Helper function, automatically sets required message fields
- Students can either use this function, or prepare message manually
- Arguments:
  - msg – message structure (allocated by caller) to be initialized
  - type – type of message
  - time – time of message sending event, unused in this laboratory work, should be 0
  - payload – pointer to payload, which will be copied to message, can be NULL (if message has empty content)
  - psize – size of payload, can be 0 for empty payload

Header file "message.h"

- Important note about size of payload
- `char s_payload[MAX_PAYLOAD_LEN];`
- `MAX_PAYLOAD_LEN` is equal to 4088, so size of s_payload is 4088 bytes

- That's a lot!
- Only useful part of payload should be sent
- fill_message() will copy exactly psize bytes from address stored by payload argument into message and set s_payload_len to value of psize

- send()/send_multicast() will send exactly s_payload_len bytes of payload

Header file "message.h"

- For this laboratory work only messages with types STARTED and DONE are used (see enumeration with types in header file)

- STARTED message
  - Should be sent by child process, when it starts execution
  - Payload should contain string, which used during printing `log_started_fmt` (see slide 19) without trailing '\0'

- DONE message
  - Should be sent by child process, when it finishes execution
  - Payload should contain string, which used during printing `log_done_fmt` (see slide 19) without trailing '\0'

Header file "log.h"

- `void shared_logger(const char * msg);`
  - Prints string to required log outputs
  - All messages should have format, described in the log.h
  - Logs will be printed to stdout and to events.log file
  - You can add your logs to stdout for debug purpose
  - **Logs to stderr are forbidden!**
  - Arguments:
    - msg – string to be printed to log outputs

Header file "log.h"

- `char * log_started_fmt;`
- `char * log_received_all_started_fmt;`
- `char * log_done_fmt;`
- `char * log_received_all_done_fmt;`
  - Log formats, which should be used with shared_logger() function
  - These events must be printed (see slides 25 and 27 with information about events)
  - First parameter of format should be 0 for this laboratory work
  - Second parameter is id of the process, which preparing the log
  - Balance parameter should be 0 for this laboratory work
  - pid X and parent pid Y should be filled with process system pid and parent system pid (not self_id!)
  - Other format strings aren't used in this laboratory work

HDU-ITMO Joint Institute
杭州电子科技大学 圣光机联合学院

Example for sending

- Let's check example of preparing and sending STARTED message by child process:

```
    // Allocate buffer for string with enough size
1   char buf[BUF_SIZE];
    // Allocate message structure on stack
2   Message msg;
    /* Prepare string with log using log_started_fmt,
     * buf – buffer to print, BUF_SIZE – maximum size
     * 1st arg is 0 in this lab, 2nd is id of process,
     * 3rd and 4th – pid and ppid (see previous slide),
     * last arg is also 0 in this lab */
3   snprintf(buf, BUF_SIZE, log_started_fmt,
             0, self_id, self_pid,
             parent_pid, 0);
```

Example for sending (continue)

- Let's check example of preparing and sending STARTED message by child process:

```
        /* Fill message with required values, where
         * STARTED – type of message
         * 0 – no timestamp in this lab
         * buf – payload to be sent
         * (string in this type of message)
         * last arg is size of useful part of payload */
4       fill_message(&msg, STARTED, 0, buf,
                     strlen(buf));
        /* Send this message to all processes
         * including parent */
5       send_multicast(&msg);
        // Print string with log_started_fmt to log
6       shared_logger(buf);
```

HDU-ITMO Joint Institute
杭州电子科技大学 圣光机联合学院

Example for receiving

- Let's check example of receiving message:

```
    // Allocate message structure on stack
1   Message msg;
2   int proc_id = 3;
    // Receive message from process with id 3
3   receive(proc_id, &msg);
    // Message is received, can use its fields
4   if (msg.s_header.s_magic == MESSAGE_MAGIC &&
5       msg.s_header.s_type == STARTED) {
6         printf("I received STARTED from "
7                "process %d with size %d\n",
8                proc_id,
9                msg.s_header.s_payload_len);
10  }
```

Task for laboratory work

- Package for preparing laboratory work has following parts:
  - libdistributedmodel.so – communication framework, presenting model of distributed system
  - labs_headers/ – directory with headers, which are presented API of communication framework
  - Makefile – file of build automation system for laboratory work to simplify local testing
  - **lab.c – template of source code file, which should be implemented by students**
  - template.docx – template for report with control questions

Task for laboratory work

- Goal: introduction to communication framework
- Task: students should implement functions parent_work() and child_work()

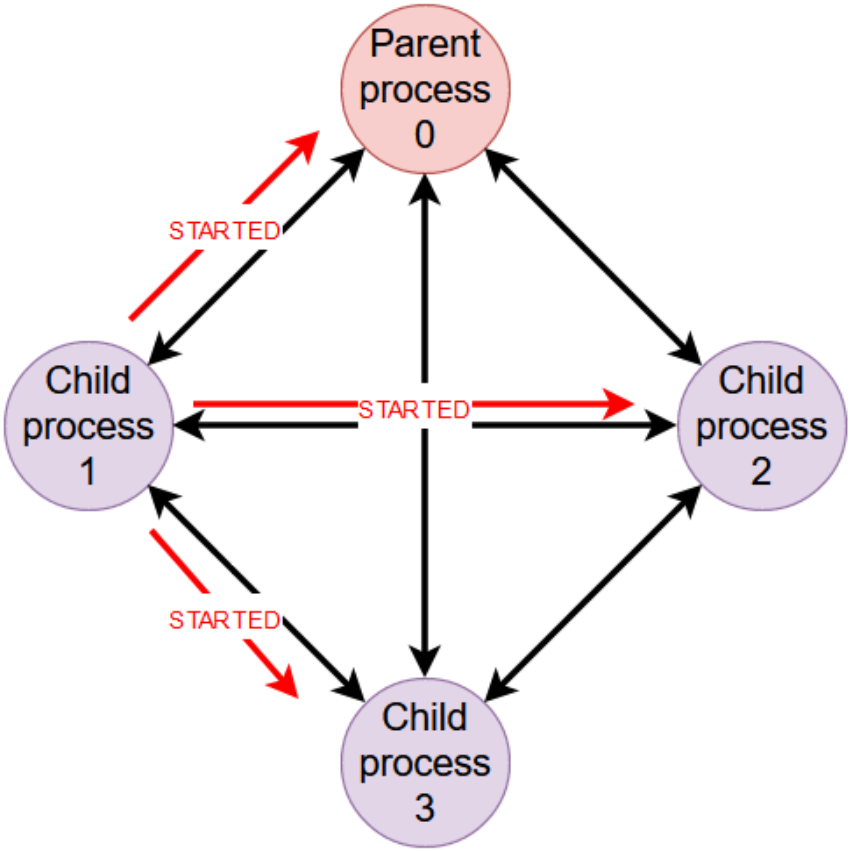
- Parent process and child processes have different workflow
- The execution of each child process consists of three subsequent phases:
  1. procedure of synchronization with all other processes in distributed system
  2. "useful" job of child process
  3. procedure of synchronization processes before their completion

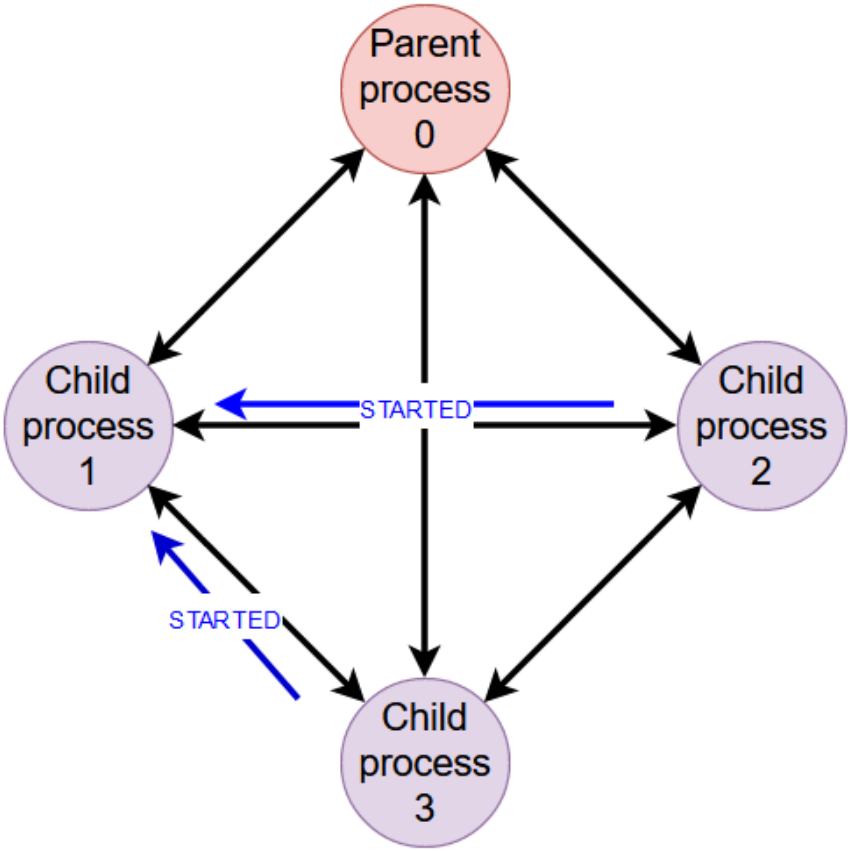
Task for laboratory work

- Phase 1 of child process
  - Process sends a message of type STARTED to all other processes, including the parent and prints log of format `log_started_fmt`
  - Process waits for STARTED messages from all other child processes and prints log of format `log_received_all_started_fmt` after all STARTED messages were received

Task for laboratory work

- Phase 1



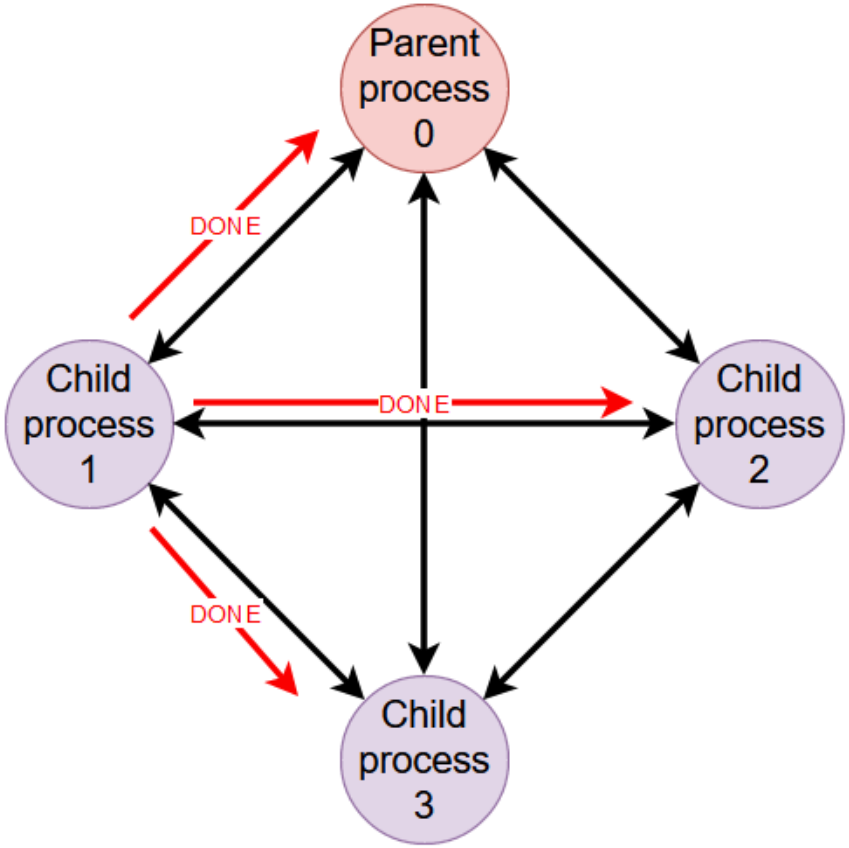


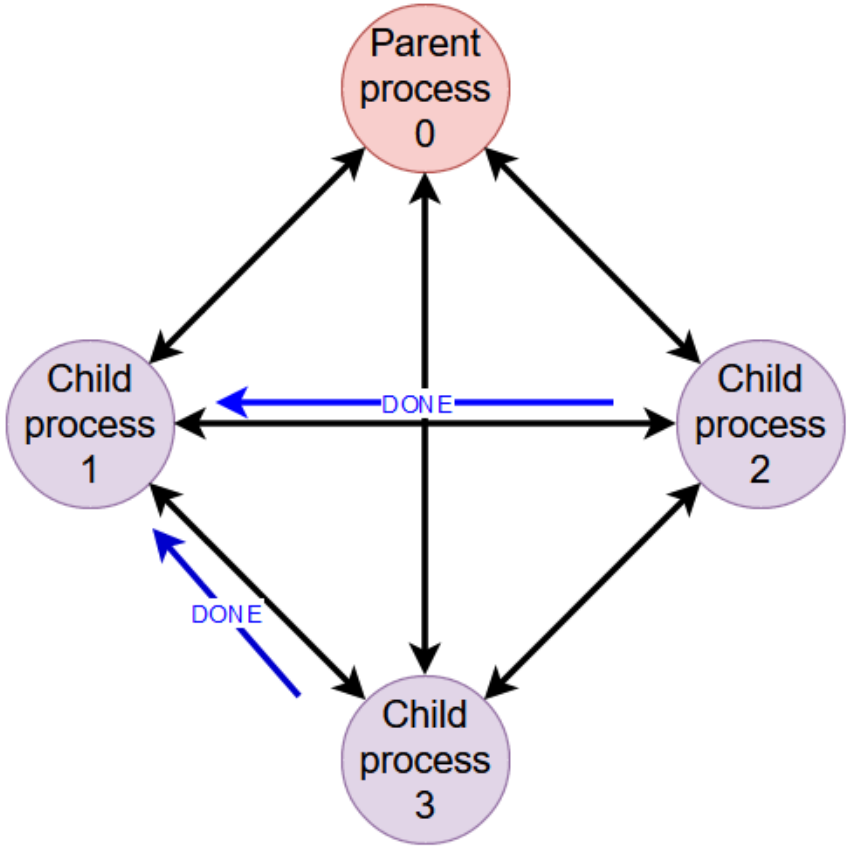
Task for laboratory work

- Phase 2
  - Process doesn't perform any "useful" work, so it immediately proceeds to the third phase of their execution
- Phase 3
  - Process sends a message of type DONE to all other processes, including the parent and prints log of format `log_done_fmt`
  - Process waits for DONE messages from all other child processes and prints log of format `log_received_all_done_fmt` after all DONE messages were received

Task for laboratory work

- Phase 3



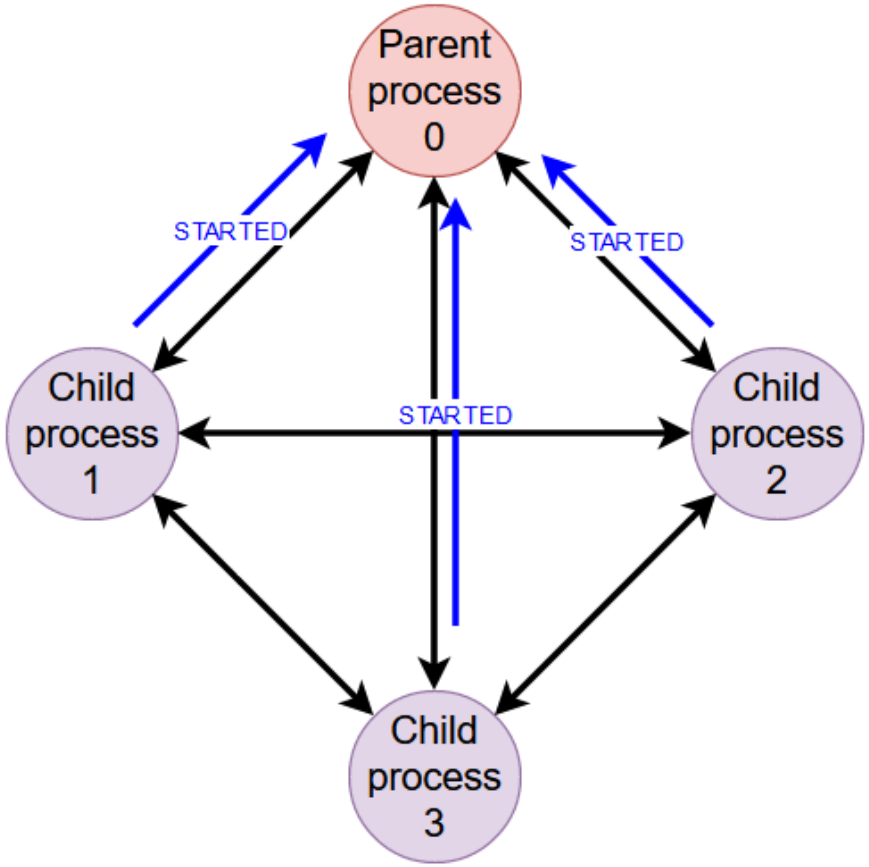


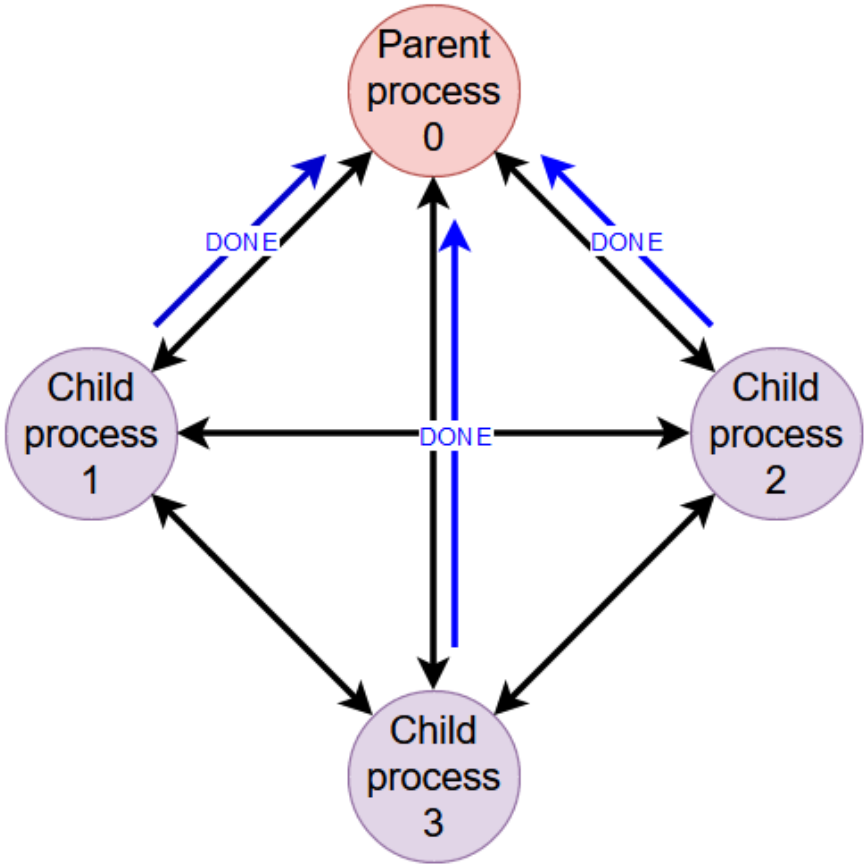
Task for laboratory work

- In this laboratory work function of parent process is limited to monitoring of child processes work
- Parent process shouldn’t send messages to child processes
- Parent process must receive all STARTED and DONE messages from child processes
- No logs are printed by parent process

Task for laboratory work

- Parent work





Compilation

- Makefile is provided with laboratory work to simplify its compilation
- Default target for make will compile "lab" binary executable file with program
- No compilation warnings are allowed
- C99

- libdistributedmodel.so supports only x64 Linux-based operating systems!
- Recommended environment for compilation:
  - Ubuntu 16.04 or newer (or other Linux-based operating system)
    - Virtual machine can be used (VMware Player, Virtual Box or other)
  - clang3.8 or newer

Execution

- To execute program, you must set environment variable LD_LIBRARY_PATH, e.g.:

```
export LD_LIBRARY_PATH="$LD_LIBRARY_PATH:/path/to/directory/with/libdistributedmodel"
```

- To execute program use following command:

```
LD_PRELOAD=/path/to/lib/libdistributedmodel.so ./lab -l 1 -p X
```

- Where "l" key – for number of laboratory work (work 1)
- And "p" key – for number of child processes [1..9]
- LD_PRELOAD environment variable should be set in mentioned way to be applied only for execution of lab binary

Bot usage

1. Pack source code to archive with the name labN.tar.gz, where N – is num of laboratory work; this archive should contain single file **lab.c**
   - Example of command, to prepare archive:
     tar czf lab1.tar.gz lab.c
   - For MacOS COPYFILE_DISABLE variable should be set:
     COPYFILE_DISABLE=1 tar czf lab1.tar.gz lab.c

2. Send mail to address ifmo.distributedclass.bot@gmail.com
   - Archive should be added to mail as attachment
   - Subject of letter should have following format (only Latin symbols):

HDU Laboratory work #N Student_Name Student_HDU_ID

Bot usage

3. If bot answers with message “Failed” – you should find the reason, fix issue and send code again
   - Also, bot can send the logs of execution with instructions, how to open them; it can be useful for debug
   - If bot doesn’t answer for 10 minutes, feel free to send message again

4. If bot answers with message “Passed” – you can prepare a report and answer on control questions

Control questions

Control questions should be answered in the report:

1. What is distributed system? Give your understanding of such systems.

2. Why starting synchronization between processes is important in this laboratory work?

3. Execute your program several times with 9 child processes. Why order of logs can be different?

Task and report

For this laboratory work students should:

1. Prepare source code for the task
2. Get reply "Passed" from the bot
3. Answer on control questions
4. Fill the report (see template.docx, all fields must be filled) and send it to dstarakanov@itmo.ru before deadline

Thank you!