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

Distributed Computing
Laboratory work #2
Distributed banking system

Michael Kosyakov
Associate Professor

Denis Tarakanov
Assistant Lecturer

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

Classes plan

1. Model of banking system
2. Communication framework API
3. Task for laboratory work 2
4. Control questions

Model of banking system

- What is banking system?
  - Customer accounts in multiple branches of the bank
  - Depositing of additional funds isn't allowed
- Problem of calculating the total amount of money at specific time:
  - Incomplete operations
  - Clock of branches can be out of sync

Model of banking system

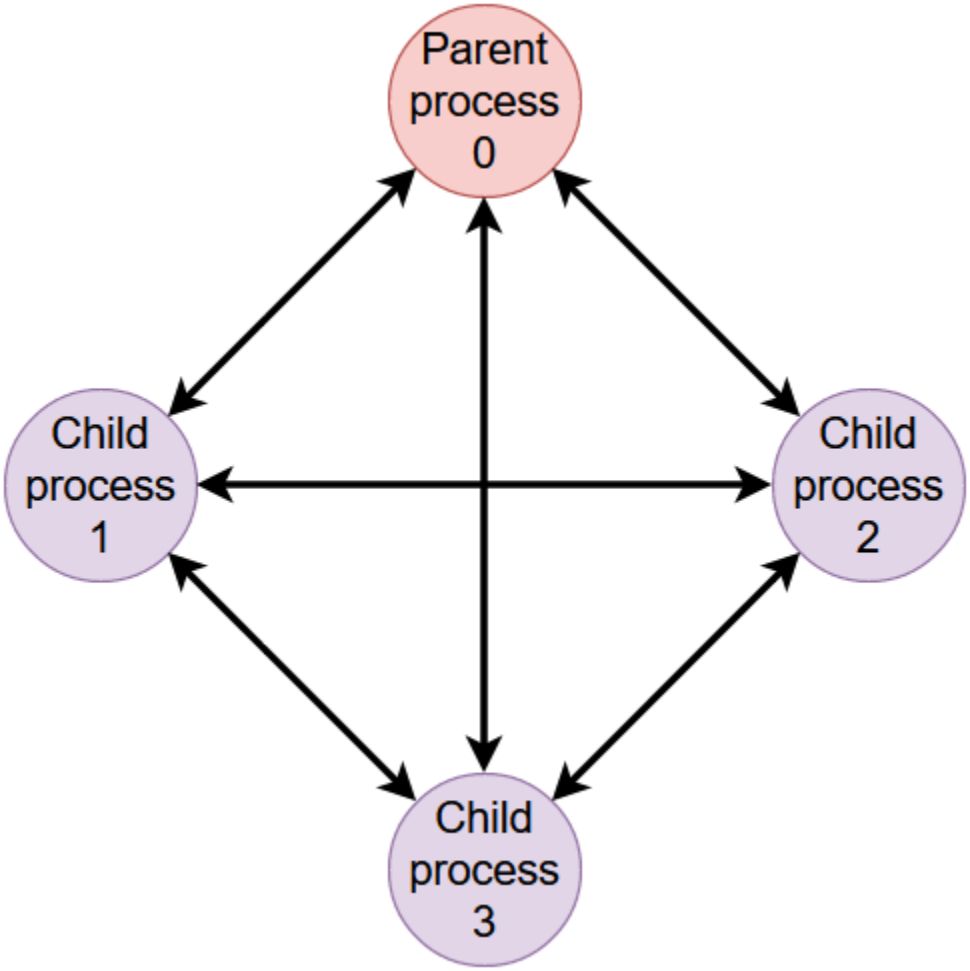
- Model from laboratory work #1 as banking system:
  - Parent process is responsible for customer's requests
  - Child processes are responsible for servicing accounts
- Two types of requests are supported:
  - Request to transfer money between accounts in different branches
  - Request to obtain information about balance history

Model of banking system

- Transfer
  - Handling of client's request
  - Transferring of money between branches
  - Client should get confirmation of result

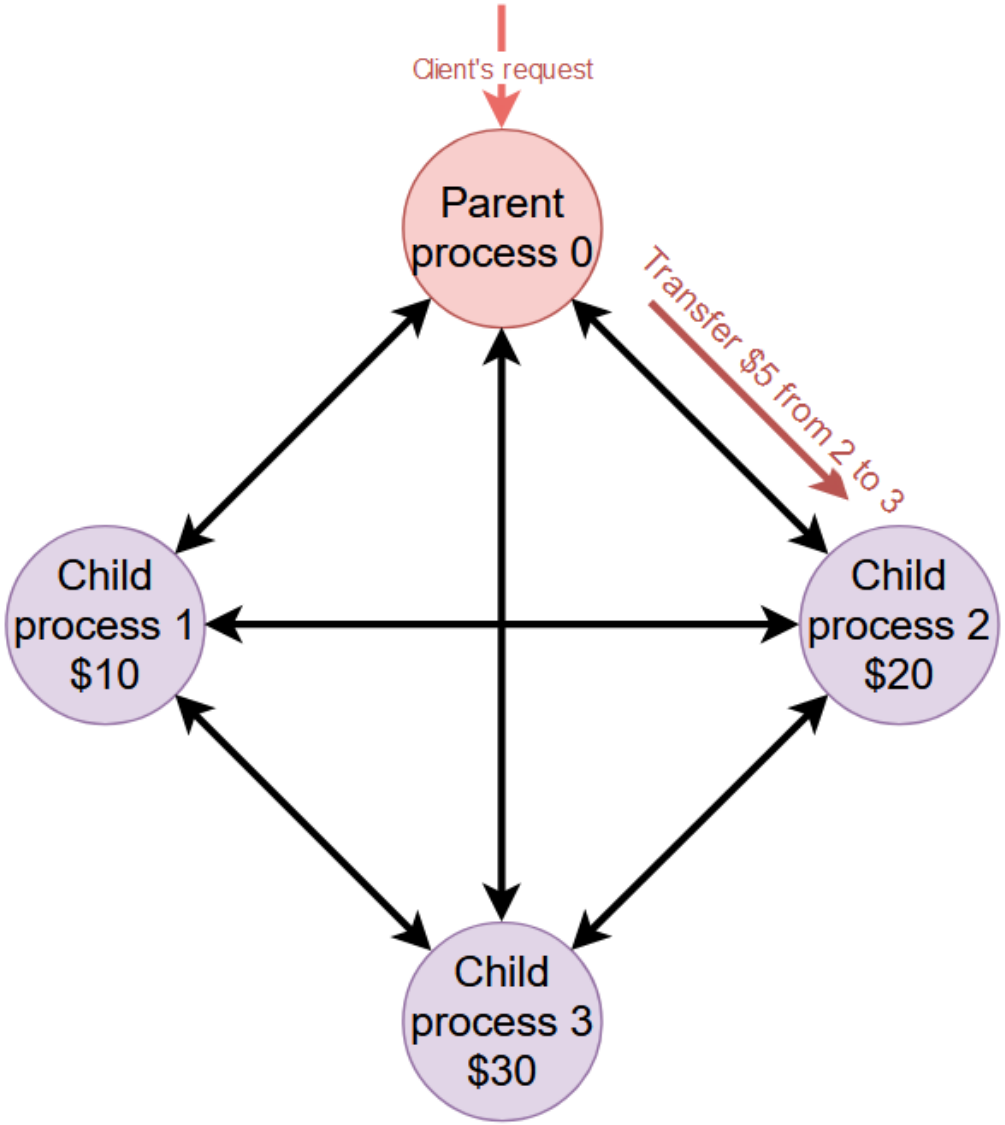
- Physical time
  - Physical clocks are “perfect” – no drifts and shifts between processes
  - Each transfer is instantaneous – clock readings don’t change since client starts transfer request and until client gets confirmation of this transfer completion

Model of banking system



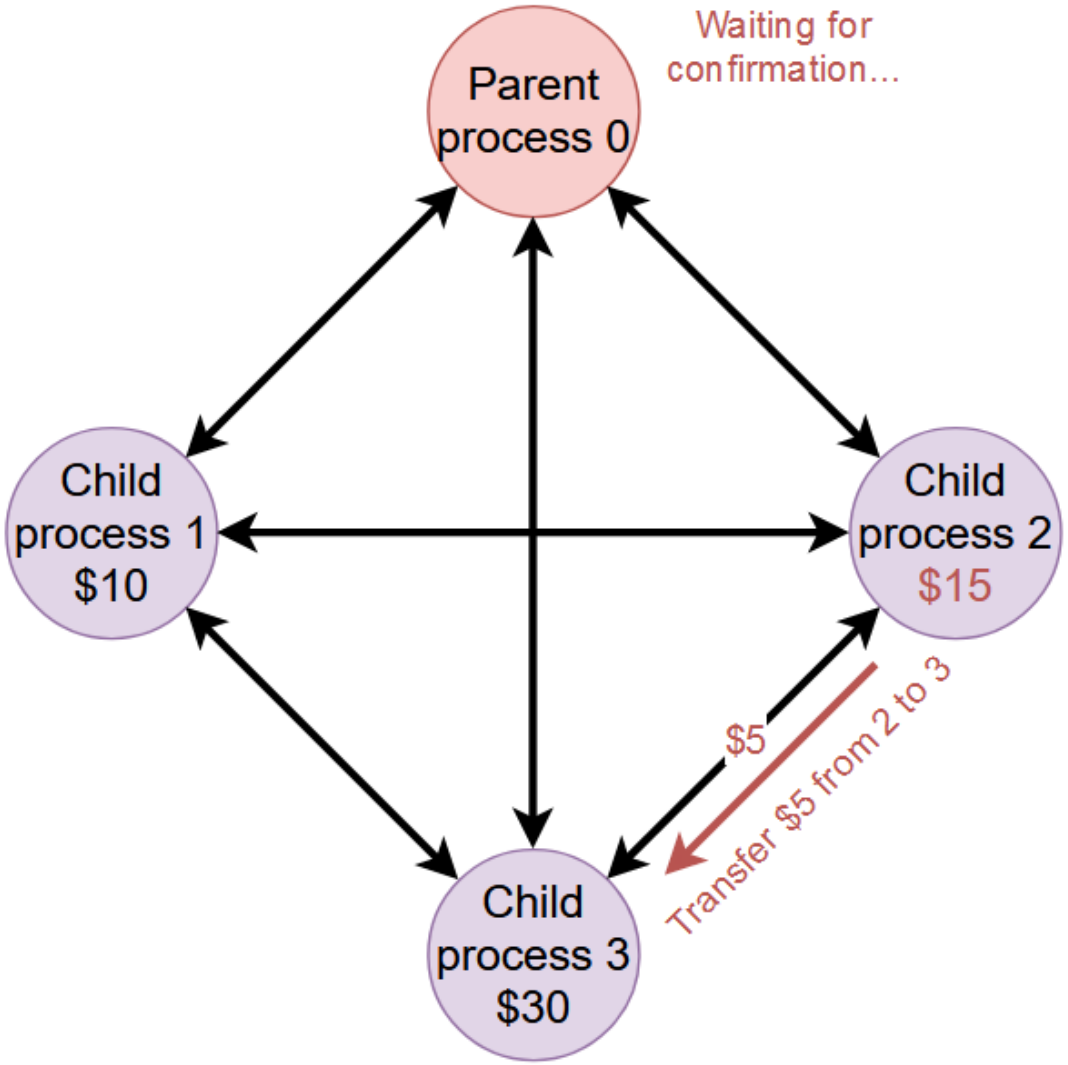
- Example for 3 child processes
- Each child process has starting amount of money
  - Process 1 has $10
  - Process 2 has $20
  - Process 3 has $30
  - Total – 60$

Model of banking system



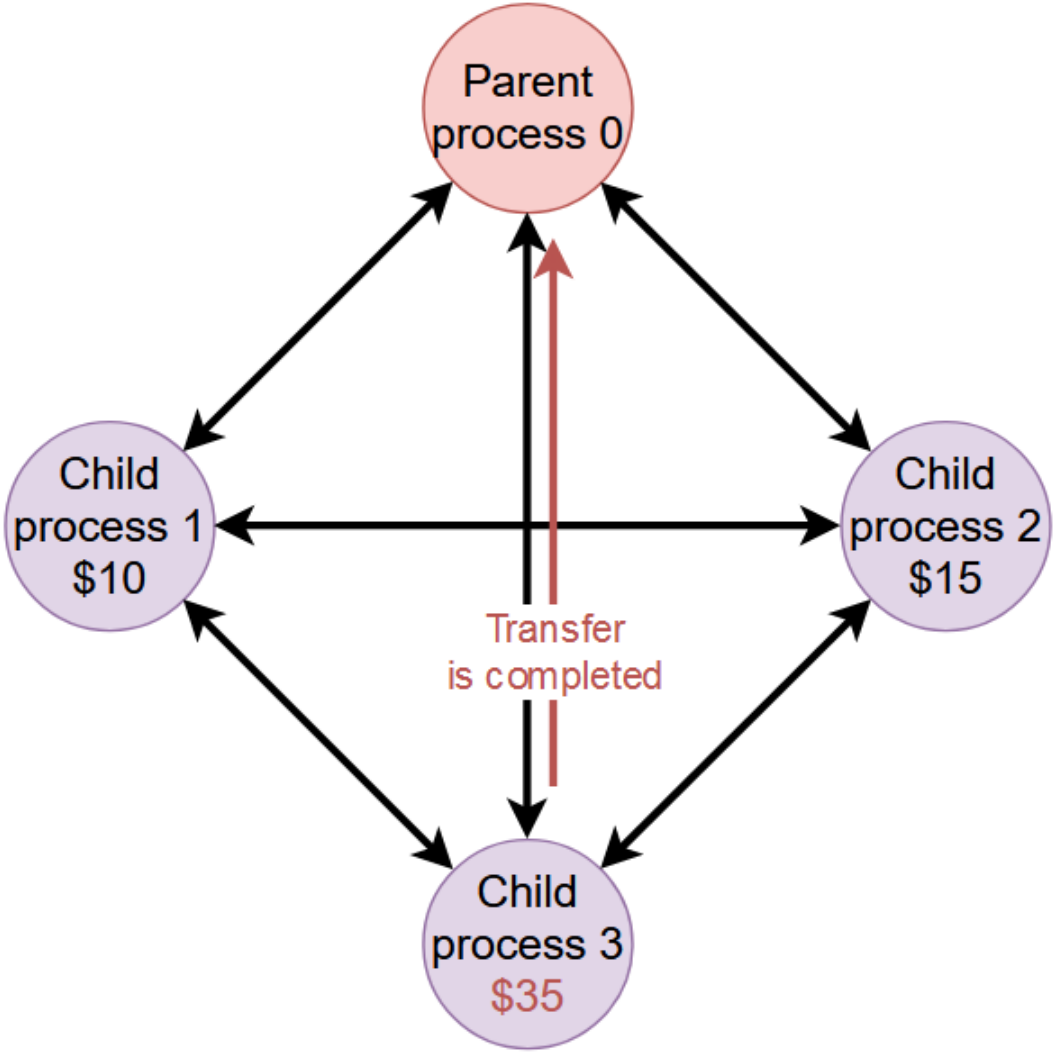
- Request for transferring $5 from process 2 to process 3
- Stage 1 – notify "source" process

Model of banking system



- Request for transferring $5 from process 2 to process 3
- Stage 2 – source process changes value on its account and notifies destination process

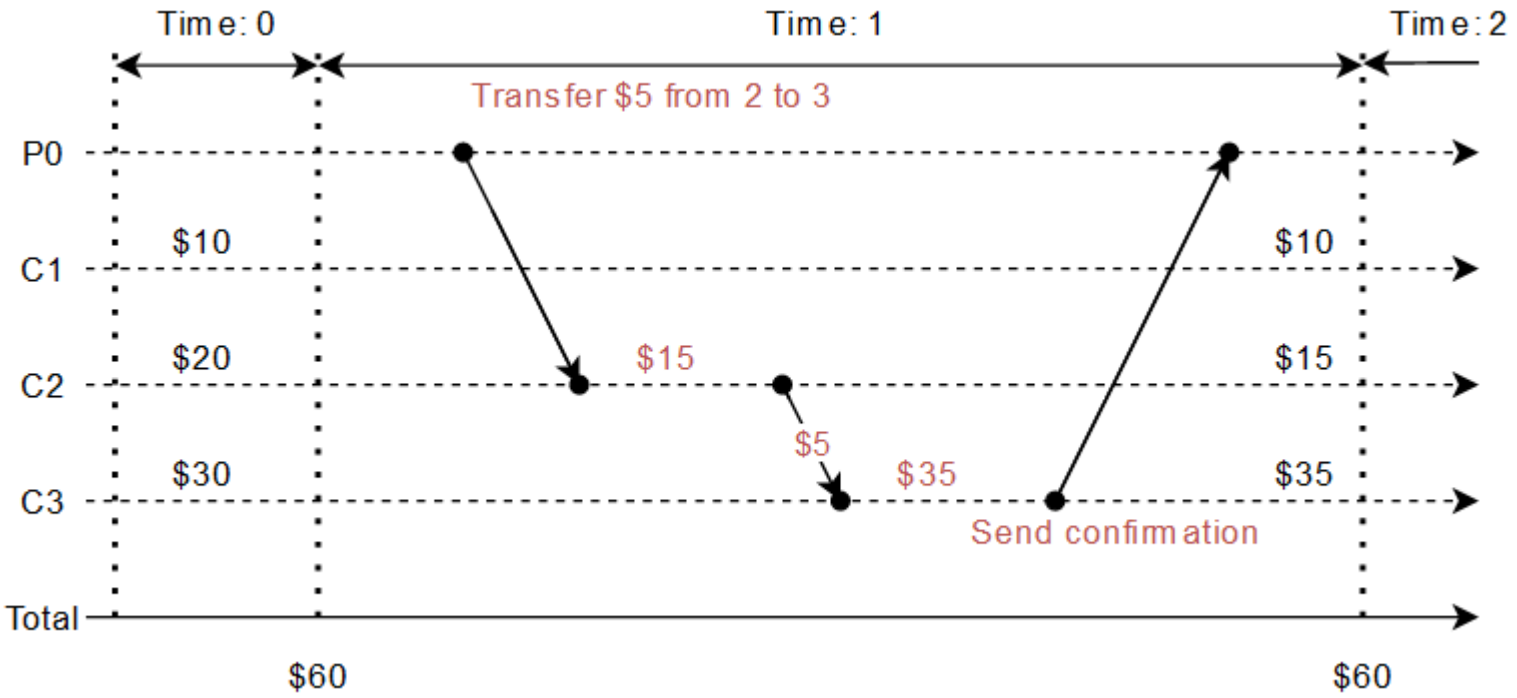
Model of banking system



- Request for transferring $5 from process 2 to process 3
- Stage 3 – destination process changes value on its account and sends confirmation to client
- After confirmation next request can be handled

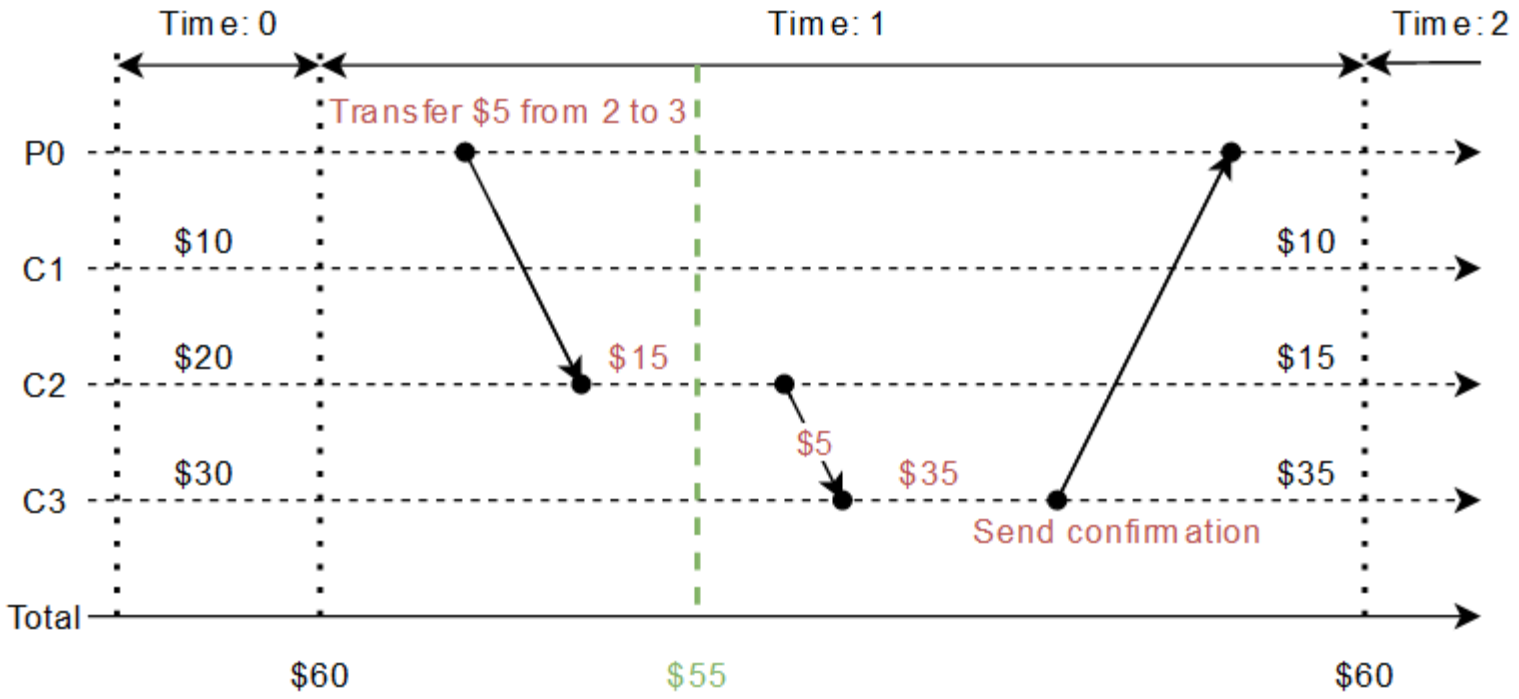
Model of banking system

- Problem of calculating the total amount of money
  - Clocks are perfect
  - Each transfer is instantaneous
  - Total amount of money is the same at any point of time



Model of banking system

- Problem of calculating the total amount of money
  - During time interval real value of money can be inconsistent
  - In the end of each time interval sum should be correct



Communication framework API

- Presented in header files from labs_headers
- Documentation for functions and structures is available as commentaries in header files

- banking.h
  - Provides signatures of functions and data structures for model of banking system. Some of them should be implemented by students
- process.h, message.h, log.h
  - Additional functions, strings and data structures should be used for banking system implementation

Header file "process.h"

```
struct child_arguments {
    local_id self_id;
    int count_nodes;
    uint8_t balance;    // New field to use
    bool mutex_usage;
};
```

- Defines required information about child process
- New field to use:
  - balance – amount of money in this child process at the start of execution
    Balance is always greater than zero

Header file "message.h"

- `int send(local_id dst,`
  `const Message * msg);`
  - Send a message to the process specified by id
  - If process with id "dst" is already terminated – framework will stop execution of program with related error message
  - Arguments:
    - dst – id of the process, which is destination of message
    - msg – pointer to message, which should be sent

Header file "message.h"

- `int receive_any(Message * msg);`
  - Receive a message from any of the processes
  - Call to this function will block execution until message from any of the processes (including parent) is received
  - Function receives the first message, which it finds in communication channels of this process, see slide 44 for example
  - Arguments:
    - msg – pointer to message structure, allocated by user, where received message should be stored
  - Return value:
    - Function returns id of process, which sent the message

Header file "message.h"

```
typedef struct {
    uint16_t      s_magic;
    uint16_t      s_payload_len;
    int16_t       s_type;
    // New field to use
    timestamp_t   s_local_time;
} __attribute__((packed)) MessageHeader;
```

- Defines a format of message header
- New field to use:
  - s_local_time – time of message sending event, in this laboratory work should be filled with timestamp of physical time, see slides 19 and 34 for more details
    Should be set, but receiver of message **shouldn't use it**

Header file "message.h"

- New messages are available for this laboratory work!

- TRANSFER message
  - Can be sent from parent to child process (source of transfer) or from one child process (source) to another child process (destination of transfer)
  - Payload should contain structure TransferOrder with information about transfer (see slide 20)

- ACK (acknowledgement) message
  - Should be sent by child process to parent process as confirmation of transfer
  - Payload should be empty; payload length should be 0

Header file "message.h"

- New messages are available for this laboratory work!

- STOP message
  - Should be sent by parent process, when there are no more transfers to be handled
  - Payload should be empty; payload length should be 0

- BALANCE_HISTORY message
  - Should be sent by child process to the parent process, after receiving messages DONE from all other child processes
  - Payload should contain structure BalanceHistory with information about processes' balance during execution (see slides 21-23)

Header file "banking.h"

- `timestamp_t get_physical_time();`
  - Get current physical time from perfect clocks
  - Implemented in a such way, that transfer happens instantaneous from point of clocks
  - Return value:
    - Current physical time of a process

Header file "banking.h"

```
typedef struct {
    local_id   s_src;
    local_id   s_dst;
    balance_t  s_amount;
} __attribute__((packed)) TransferOrder;
```

- Defines a format of message payload with information about transfer
- Fields:
  - s_src – id of child process, which is source of transfer
  - s_dst – id of child process, which is destination of transfer
  - s_amount – amount of money to be transferred from source process to destination

Header file "banking.h"

```
typedef struct {
        balance_t   s_balance;
        timestamp_t s_time;
        balance_t   s_balance_pending_in;
} __attribute__((packed)) BalanceState;
```

- Defines a format for storing information of process's balance at specific time
- Fields :
  - s_balance – balance of process at time s_time
  - s_time – specified time for this balance (should be equal to get_physical_time() return value)
  - s_balance_pending_in – field is unused in this laboratory work, should be always equal to 0

Header file "banking.h"

```
typedef struct {
    local_id        s_id;
    uint8_t         s_history_len;
    BalanceState    s_history[MAX_T + 1];
} __attribute__((packed)) BalanceHistory;
```

- Defines a format for storing information of process's balance during execution and format of payload for BALANCE_HISTORY message
- Fields :
  - s_id – id of this child process for identification
  - s_history_len – amount of useful BalanceStates in s_history (**index of last filled BalanceState + 1**)
  - s_history – array of all BalanceStates of this process s_history[N] is BalanceState at time N

Header file "banking.h"

```
typedef struct {
    uint8_t          s_history_len;
    BalanceHistory   s_history[MAX_PROCESS_ID + 1];
} AllHistory;
```

- Defines a format of collecting the whole history for all child processes
- Fields :
  - s_history_len – number of BalanceHistories (exactly number of child processes)
  - s_history – array of BalanceHistories (one from each process)
    s_history[N] is BalanceHistory of process with id (N+1)
- See slide 28 for details about histories

Header file "banking.h"

- `void bank_operations(local_id max_id);`
  - Main banking function, imitates client requests for money transfers
  - Should be called by parent process (see slide 40)
  - Implementation of this function is already presented in lab.c and **shouldn't be changed** by student
  - Real test implementation can be different
  - Arguments:
    - max_id – maximum identifier of child processes (e.g., if you have 3 child processes, max_id should be equal to 3)

Header file "banking.h"

- `void transfer(local_id src, local_id dst, balance_t amount);`
  - Implementation of transfer operation
  - Should be implemented by students
  - Called by bank_operations() function
  - Arguments:
    - src – id of child process, which is source of transferring
    - dst – id of child process, which is destination of transferring
    - amount – amount of money, which should be transferred

Header file "banking.h"

- `void transfer(local_id src, local_id dst, balance_t amount);`

- During transfer parent process should:
  1. Prepare TransferOrder
  2. Send TRANSFER message to source child process
  3. Wait for ACK message from destination child process

- After these operations transfer is completed

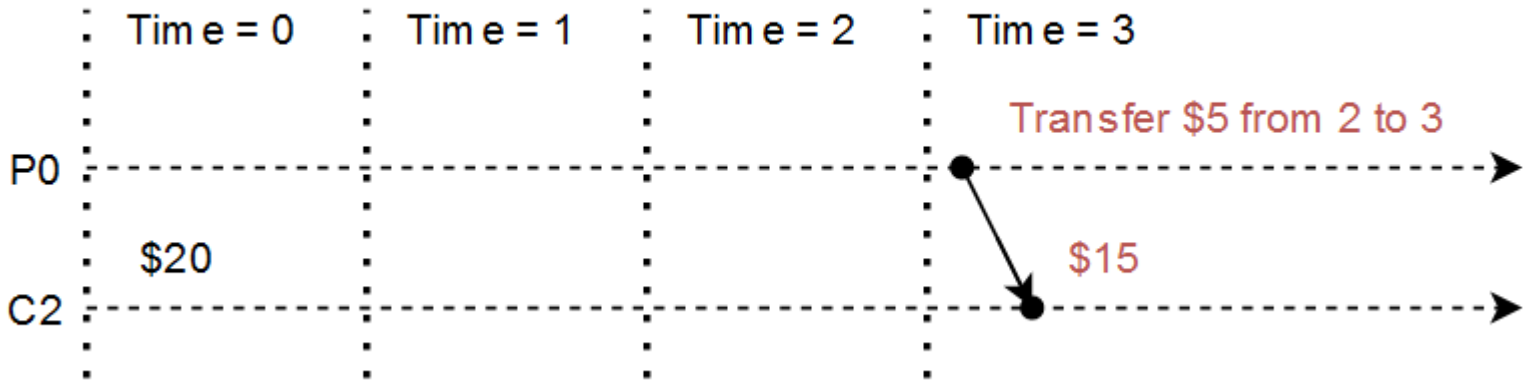
Header file "banking.h"

```
void print_history(
        const AllHistory * history);
```

- Prints BalanceHistories to stdout
- See slide 36 with explanations, how AllHistory should be prepared
- Arguments:
  - history – pointer to AllHistory structure, which contains information of BalanceHistories of each child process

Header file "banking.h"

- "Fixing" BalanceHistory
- BalanceHistory should be updated for each moment of time
- Let's check following situation
  - At moment t = 0 balance of process C2 is $20
  - At moment t = 3 process C2 receives TRANSFER message to transfer $5 to process 3



Header file "banking.h"

- "Fixing" BalanceHistory
- BalanceHistory should be updated for each moment of time
- Let's check following situation
  - At moment t = 0 balance of process C2 is $20
  - At moment t = 3 process C2 gets TRANSFER message to transfer $5 to process 3
  - s_history_len should be updated to 4, because there are 4 useful values in BalanceHistory!

current time

Time	0	1	2	3	4
C2 balance $	20	0	0	0	0

Time	0	1	2	3	4
C2 balance $	20	?	?	**15**	0

Header file "banking.h"

- "Fixing" BalanceHistory
- BalanceHistory should be updated for each moment of time
- Let's check following situation
  - At moment t = 0 balance of process C2 is $20
  - At moment t = 3 process C2 gets TRANSFER message to transfer $5 to process 3
  - s_history_len should be updated to 4, because there are 4 useful values in BalanceHistory!

Time	0	1	2	3	4
C2 balance $	20	?	?	15	0

Time	0	1	2	3	4
C2 balance $	20	**20**	**20**	15	0

Header file "banking.h"

- "Fixing" BalanceHistory
- BalanceHistory should be updated for each moment of time
- Let's check following situation
  - At moment t = 0 balance of process C2 is $20
  - At moment t = 3 process C2 gets TRANSFER message to transfer $5 to process 3

Time	0	1	2	3	4
C2 balance $	20	**20**	**20**	15	0

- print_history() will try to do it for you
  - If it detects 0 in history between 2 values different than 0
  - But it is better to fix it by yourself during update
- Correct s_history_len should be set by students

Header file "log.h"

- Important changes to log formats
  - First parameter of formats should be current physical time of the process: get_physical_time() should be used
  - Balance parameter should be equal to the balance of the process:
    - Initial balance for log_started_fmt
    - Final balance for log_done_fmt

Header file "log.h"

- `char * log_transfer_out_fmt;`
- `char * log_transfer_in_fmt;`
  - New log formats, which should be used with shared_logger() function
  - These events must be printed, when child process handles TRANSFER message (first format, if message was sent from parent process, second format, if message was sent from another child process)
  - First parameter of formats should be current physical time
  - Second parameter is id of the process, which preparing the log
  - Third parameter – is amount of transferring money
  - Fourth parameter – is either for id of destination process, or id of source process

Example

- Example of preparing empty message (STOP message)

```
    // Allocate message structure on stack
1   Message msg;
    // Get current physical time
2   timestamp_t current = get_physical_time();
    // Prepare a message, payload is NULL, length is 0
3   fill_message(&msg, STOP, current, NULL, 0);
```

Example

- Send structure BalanceHistory with name “history”

```
    // Allocate message structure on stack
1   Message msg;
    // Get current physical time
2   timestamp_t current = get_physical_time();
    /* Size of useful part of BalanceHistory – 2 technical
     * fields and s_history_len structures BalanceState */
3   uint16_t psize = 2 * sizeof(uint8_t) +
                     history.s_history_len *
                     sizeof(BalanceState);
    // Prepare a message, payload is NULL, length is 0
4   fill_message(&msg, BALANCE_HISTORY, current,
                 &history, psize);
```

Example

- Receive structure BalanceHistory

```
    // Allocate structure AllHistory on stack
1   AllHistory all_history;
    // Allocate message structure on stack
2   Message msg;
    // Receive the structure from 2 (or use receive_any())
3   receive(2, &msg);
    // Interpret payload as BalanceHistory structure
4   BalanceHistory * history =
                (BalanceHistory *)msg.s_payload;
    // Copy BalanceHistory to all_history on position (2-1)
5   memcpy(&all_history.s_history[1], history,
            msg.s_header.s_payload_len);
```

Example

- Example of typical usage of receive_any()

```
    // Allocate message structure on stack
1   Message msg;
    // e.g., have received all DONE messages
2   while(out_condition) {
        // Interpret payload as BalanceHistory structure
3       receive_any(&msg);
        // Check, what is type of message and handle it
4       switch (msg.s_header.s_type) {
5           case MESSAGE_TYPE1:
6               <…>
7           case MESSAGE_TYPE2:
8               <…>
9       }
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: implement banking system
- Task: students should implement functions parent_work(), child_work() and transfer()

- 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

- Parent process workflow
  1. Parent process must receive all STARTED messages from child processes
  2. Parent process should call function bank_operations(), which uses transfer() to send messages to child processes
     See slides 25-26 with information, how transfer() should be implemented
  3. After execution of bank_operations() parent process sends STOP message to all child processes
  4. Parent process must receive all DONE messages from child processes
  5. Parent process must receive all BALANCE_HISTORIES messages from child processes, aggregate their payload to AllHistory and call print_history()

- No logs are printed by parent process

Task for laboratory work

- Phase 1 of child process (same as in laboratory work #1)
  - 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

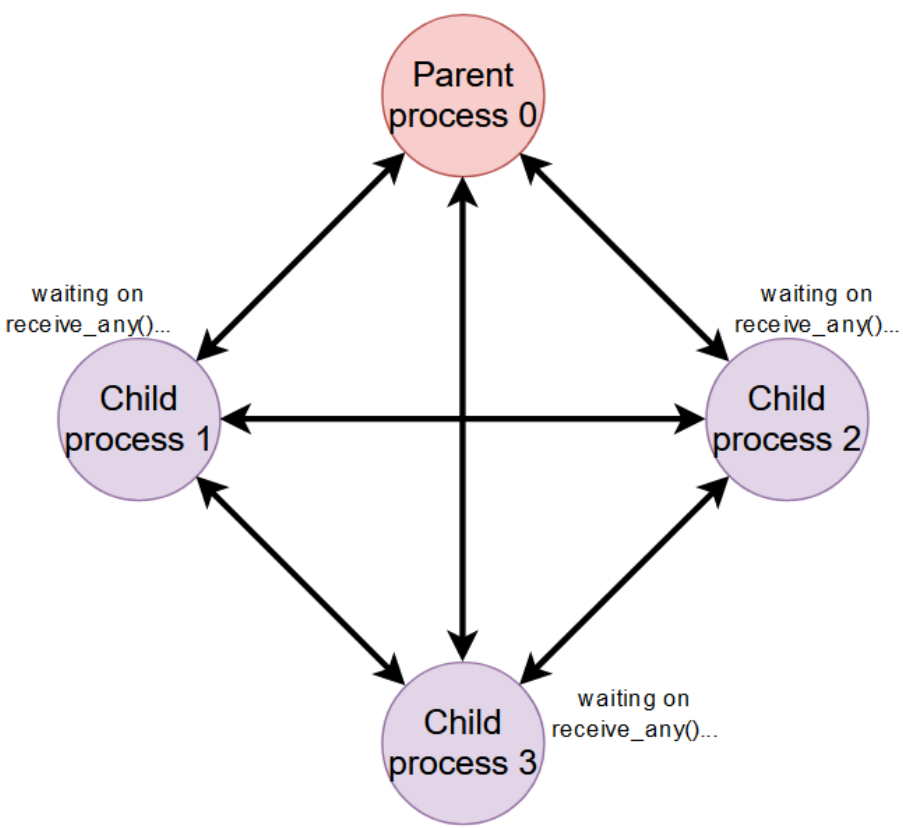
- Phase 2 of child process
  - Useful work of child process is waiting and handling TRANSFER and STOP messages
- TRANSFER message
  - If child process is source of transfer – it should update its balance and BalanceHistory (including s_history_len) and then send this TRANSFER message to destination of transfer
  - If child process is destination of transfer – it should update its balance and BalanceHistory (including s_history_len) and then send ACK message to parent process to complete transfer
  - Update balance and BalanceHistory before sending messages!
  - Use get_physical_time() to obtain current time
- STOP message
  - Child process goes to the Phase 3

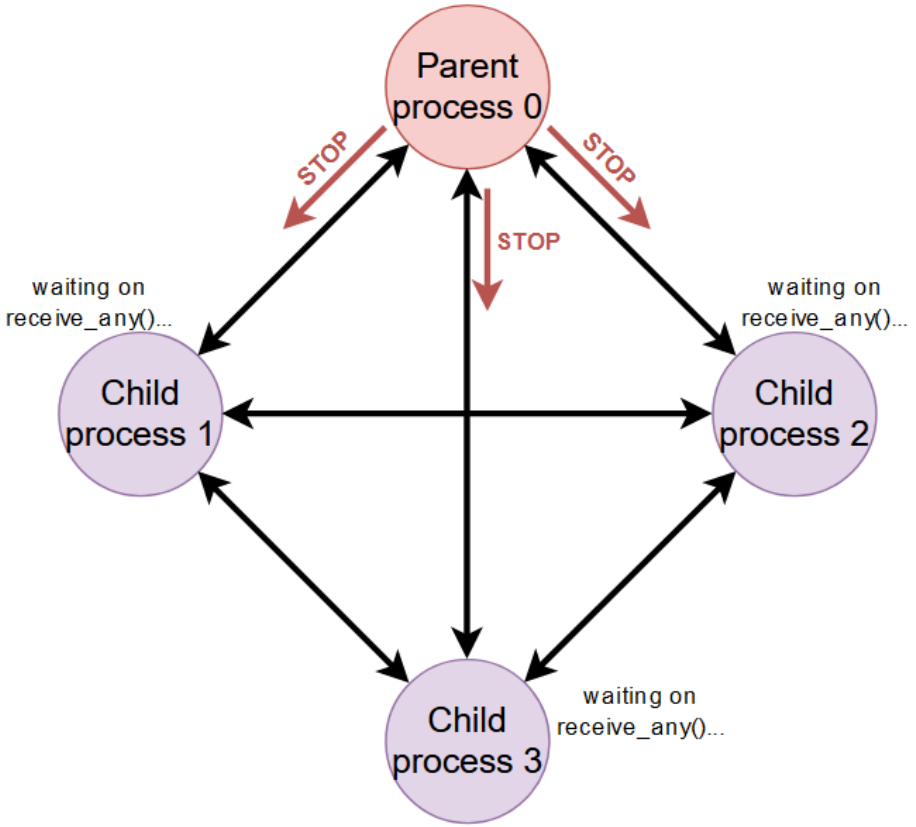
Task for laboratory work

- Phase 3 of child process
  (first part is the same as laboratory work #1)
  - 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
  - After that process should prepare BalanceHistory structure and send it as payload for BALANCE_HISTORY message to parent process

Task for laboratory work

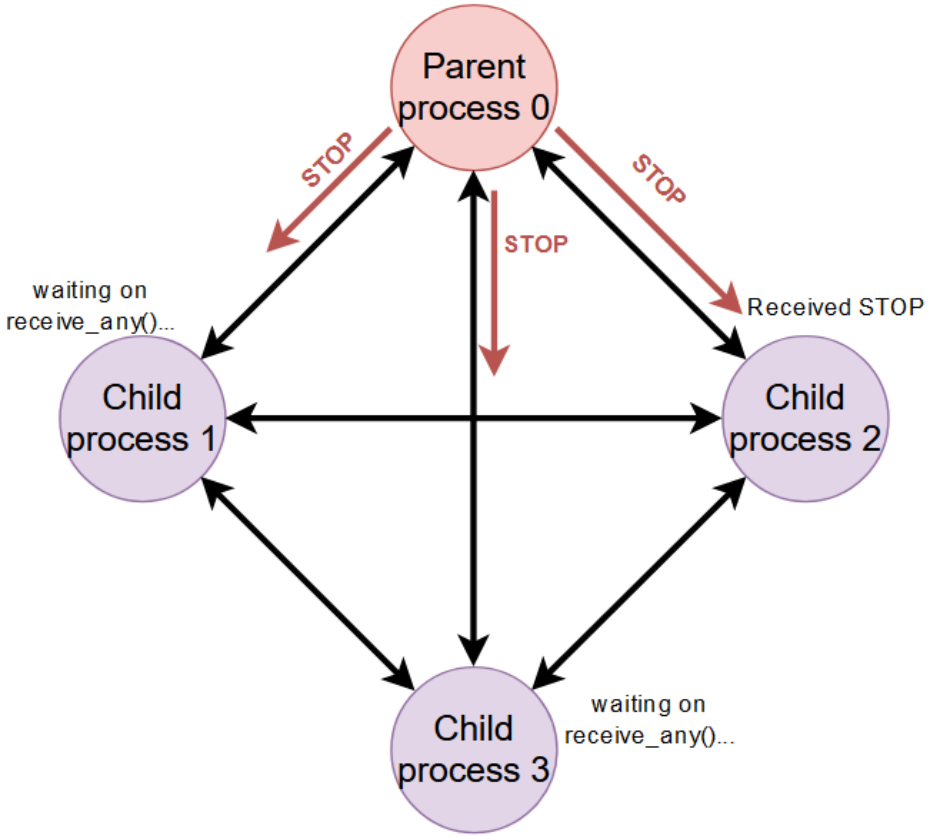
- Using receive_any() in Phase 2

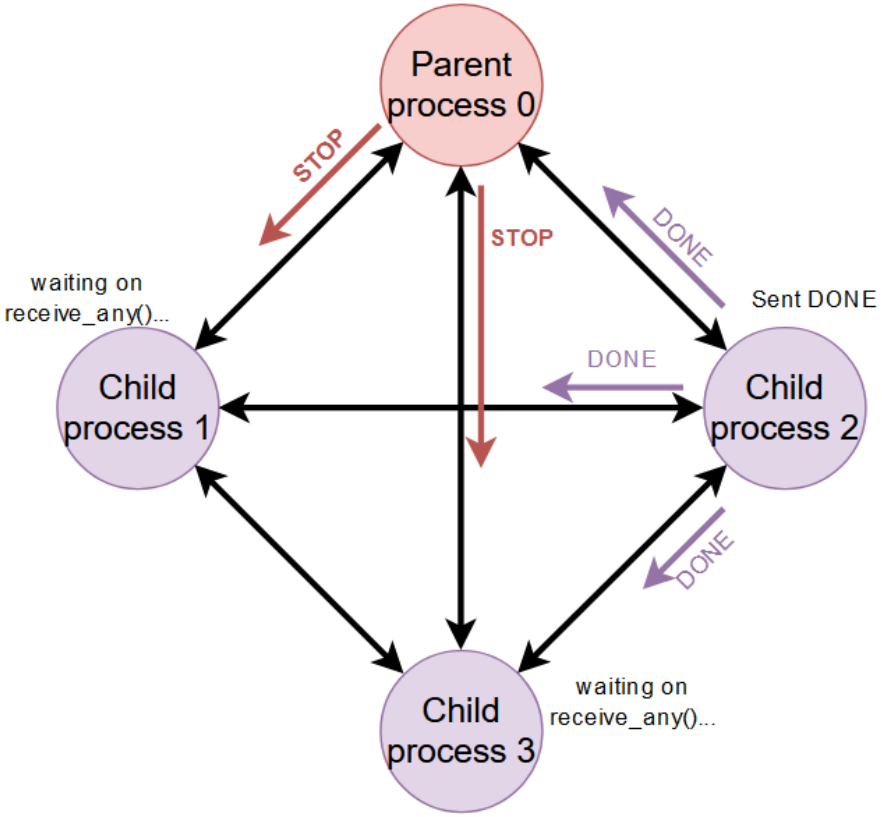




Task for laboratory work

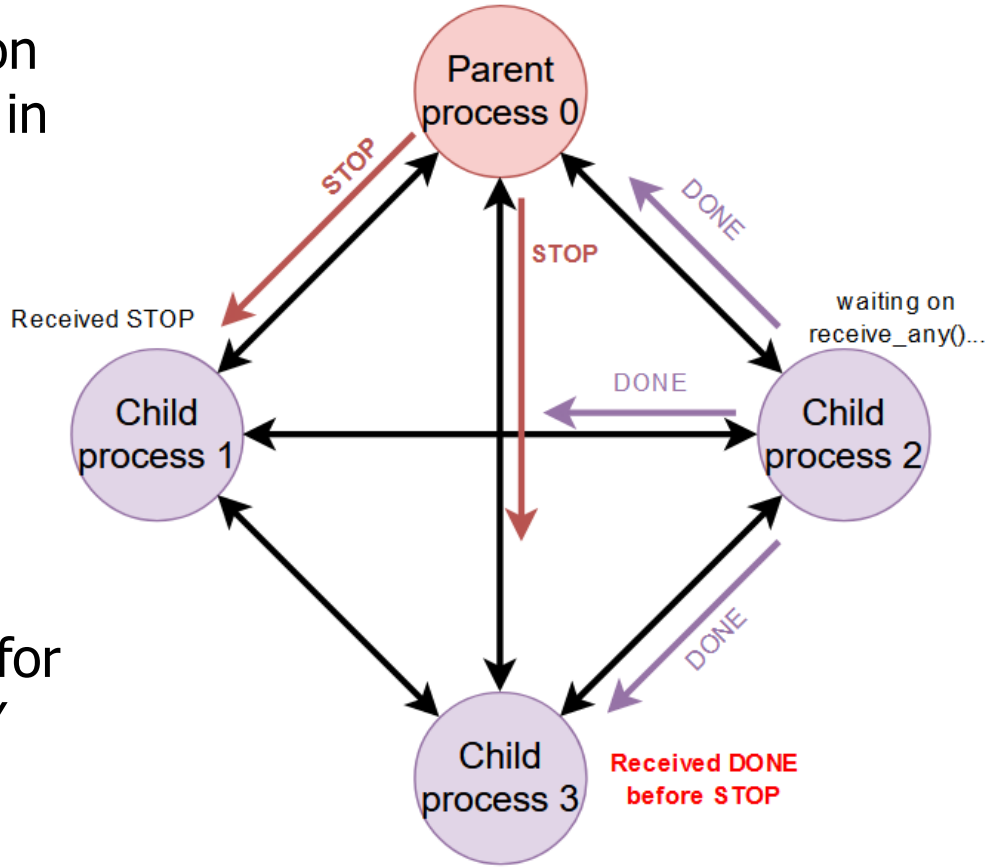
- Using receive_any() in Phase 2





Task for laboratory work

- Using receive_any() in Phase 2
  - receive_any() result depends on which channel will be checked in that moment of time
  - In this situation child process should expect not only TRANSFER and STOP type of messages, but also DONE message
  - The same can happen with parent process during waiting for DONE and BALANCE_HISTORY messages, if you will use receive_any()



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 2 -p X B1 B2 <..> BX
```

- Where "l" key – for number of laboratory work (work 2)
- And "p" key – for number of child processes [1..9]
- B1, B2, <…> BX – is starting balances of child processes, for example, for 5 child processes:

```
./lab -l 2 -p 5 10 20 30 40 90
```

- 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 **lab2.tar.gz** lab.c
   - For MacOS COPYFILE_DISABLE variable should be set:
     COPYFILE_DISABLE=1 tar czf **lab2.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. Transfer request. Is it synchronous or asynchronous in our model from point of client? Why?
   What is difference between synchronous and asynchronous operations?

2. What will happened, if transfer operations isn't instantaneous?
   Can we calculate total amount of money at each moment of time? Why?

Control questions

Control questions should be answered in the report:

3. Make an experiment. You can find in header banking.h function get_physical_time_skew(). Use it instead of get_physical_time(). This function provides non-synchronized clocks. Why result of execution is different?
   Answer should contain output table.

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!