# PART 1

**Section 1 (Task 1):**

## Data Structure:

- KThread object; to store which thread is calling join
- Boolean flag; to prevent calling join second time on the same thread

## Changes and Implementation:

KThread.join():

- Checked if the joining thread has finished
- If not then Disabled interrupt
- Checked the flag if the thread can be joined
- Calling thread = current thread
- Made the calling thread sleep
- Restored interrupt

KThread.finish():

- Disabled interrupt
- Set the thread's status to finished
- Checked if any thread called join on the thread
- Made the calling thread ready again
- Terminated the thread

## Testing:

Kthread.selftest():

- New jointest object is passed to test the join

kthread.jointest():

- A new pingtest object is passed to create new thread
- kthread.fork() is called
- kthread.join() is called

**Section 2 (Task 2:)**

## Data Structure:

- Lock object
- Queue; to store waiting threads

## Changes and Implementation:

Condition2.constructor():

- Queue initialized in the constructor

Condition2.sleep():
- Disabled interrupt before releasing lock
- Released lock
- Made the thread sleep
- Acquired lock
- Restored interrupt after acquiring lock

Condition2.wake():
- Disabled interrupt
- Made the first thread in the waiting queue ready (if not null)
- Restored interrupt

Condition2.wakeAll():
- Disabled interrupt
- Made all threads in the waiting queue ready
- Restored interrupt

Testing:
We called wake() and sleep() methods in communicator class and as a result, its accuracy will be auto-tested during task 4 testing.

**Section 3 (Task 3):**

Data Structure:
- Pair object; to contain a thread and its corresponding waiting time
- Queue; to store the waiting threads

Changes and Implementation:

Alarm.waitUntil():
- Acquired the current thread
- Calculated its waking time
- Disabled interrupt
- Add the thread with its waking time to the queue
- Made the thread sleep
- Restored interrupt

Alarm.timerInterrupt():
- Disabled interrupt
- looped on the queue and removed one by one

- checked if a thread's wakeup time is up
- If yes, made that thread ready
- Otherwise added that thread again to the queue
- Restored interrupt

Testing:

Alarm.selfTest():
- A New AlarmTest object is passed in the new kthread constructor to test the tasks.
- Kthreads are forked and joined

Alarm.AlarmTest():
- Passed waiting time and threadedKernel.alarm into its constructor
- The current thread at the current time is shown
- The current thread is called to wait until a specified time
- Checking current thread wake time

**Section 4 (Task 4):**

Data Structure:
- Lock object
- speakerVariable (condition2 object)
- listenerVariable (condition2 object)
- Speaker flag
- Listener flag

Changes and Implementation:

Communicator.speak():
- Acquired lock
- Checked if speaker can speak ; if not then made the speaker thread sleep
- Set the speaker flag true which means one speaker is already speaking
- Set the listener flag false which means one listener can listen now
- Stored the data
- made all listeners wake up
- Made the current current speaker thread sleep and set the speaker flag false which means another speaker can speak now
- Set the listener flag true which mean one listener is already listening
- Made all speakers wake up
- Released lock

Communicator.listen():
- Acquired lock

- Checked if a listener can listening; if not made the listener thread sleep
- Set listener flag true which means one listener is already listening
- Got  the data
- made all speaker wake up
- Released lock

Testing:
(We tested task 2 and task 4 together)

Condition2.selfTest():
- New Condition2Test object is passed

Condition2.Contion2Test():
- Created some listener and speaker threads, forked them and joined them.

Condition2.Speaker():
- Communicator object is passed into the constructor
- Message is created and passed through communicator's speak function

Condition2.Listener():
- Communicator object is passed into the constructor
- Message is listened though communicator's listen function

# PART 2

**Section 1 (Task 1):**

Data Structure:
- File list ; to contain file descriptors
- Process ID
- Lock ; to make increment or decrement operation atomic

Changes and Implementation:

UserProcess.readVirtualMemory():
- Calculated the end virtual address
- Stored the main memory in a byte array
- Checked if the start and end virtual addresses are valid
- Looped from first virtual page number to last virtual page number
  - found the start and end virtual addresses of the page
  - calculated the amount can be read from the page

- found the readOffset (offset is the amount to jump from the start virtual address to find the required virtual address)
- calculated the physical start address using the pageTable and the physical page number and readOffset
- copied the memory content to data array using the amount to read
- Added the read amount to current start virtual address to find the next start virtual address

UserProcess.writeVirtualMemory():

- Calculated the end virtual address
- Stored the main memory in a byte array
- Checked if the start and end virtual addresses are valid
- Looped from first virtual page number to last virtual page number
  - found the start and end virtual addresses of the page
  - calculated the amount can be written to the page
  - found the writeOffset
  - calculated the physical start address using the pageTable and the physical page number and writeOffset
  - copied the data content to the byte array (which stores main memory)
  - Added the write amount to current start virtual address to find the next start virtual address

UserProcess.handleRead():

- Checked if file descriptor and the virtual address is valid
- Checked if the file descriptor is 0 which refers to standard input
- Stored console input in a buffer
- Wrote buffer content into main memory using writeVirtualMemory

UserProcess.handleWrite():

- Checked if file descriptor and the virtual address is valid
- Read main memory in a buffer using readVirtualMemory
- Checked if the file descriptor is 1 which refers to standard output
- Wrote buffer content to console

UserProcess.handleHalt():

- Checked if the process is the root process; if yes then halted the system

## Testing:

We checked if

- ‘printf’ can write in the console and
- ‘readline’ can take input from the user

and they worked.

**Section 2 (Task 2):**

## Data Structure:

UserKernel():

- Linked list ; to keep track of the physical page numbers of the pages can be used

UserProcess():

- Integer variable (numPages) ; to store number of contiguous pages occupied by the program
- Linked list ; to contain children
- Hashmap ;  to store children's exit status
- Integer variable(currently running) ;  to keep track of how many processes are running

## Changes and Implementation:

UserKernel:
initialize():

- Added physical page numbers to the list

allocatePage():

- Remove first page number from the list

reclaimPage():

- Added the page number of the previously used but now empty page to the front of the list

UserProcess:
execute():

- Acquired lock
- Increased the count of currently running processes to 1
- Released lock

allocate():

- Kept a local arraylist to store the used pages
- Looped on totalPage to allocate
  - checked if the starting virtual address of the page is valid
  - found a physical page number using allocatePage function from UserKernel

- if the physical page is already allocated, decremented the number of pages already used and returned all the already allocated pages to main memory and returned false
- otherwise if the page is free, added it to local ArrayList and increased the number of pages already used and returned true

returnAllPages():

- If all the required sections can not be loaded, then returned already allocated valid pages to the main memory

load():

- Checked if all pages for all the sections can be allocated
- If can not be allocated, return all pages to main memory and set the number of pages already used to 0
- Checked if pages can be allocated for stack
- If can not be allocated, return the pages to main memory
- Checked if a page can be allocated for all the arguments
- If can not be allocated, return that page to main memory

loadSections():

- Checked if the TranslationEntry for the sections are valid
- Loaded the page using physical page number

unloadSections():

- Returned all pages to main memory
- set the number of pages already used to 0
- Closed all the non-null file descriptors and made them null
- Closed the executable coff file

Testing:

- Tested after we had done Task 3

**Section 3 (Task 3):**

Data Structure:

- Lock (parentAccessLock) ; to prevent more than one child to access parent's member variables at a time
- LinkedList of children processes
- Hashmap of the status of children processes
- Uthread object ; used as user-level thread

Changes and Implementation:

UserProcess.handleExec():

- Calculated the last virtual address

- Checked if the starting virtual address of the file and arguments, total number of arguments are valid
- Checked if the filename is valid
- Copied arguments using readVirtualMemoryString to a string array
- Created a child process
- Checked if the child process can be executed
- Added this child to the LinkedList of children and set the parent of the child process

UserProcess.handleJoin():
- Checked if the processID and virtual address of thread status is valid
- Found the process on which join is called from the children LinkedList
- Called join on the child thread
- Acquired parentAccessLock
- Found the status of this child process from the status Hashmap
- Removed this child process from both children arraylist and status hashmap
- Released parentAccessLock
- Checked if the status can be stored in the main memory

UserProcess.handleExit():
- Checked if this thread's parents is null
- Acquired parentAccessLock
- Set the child status of this thread's parent's status hashmap
- Released parentAccessLock
- Unloaded the sections
- Made all the childrens' parent null
- Acquired lock
- Decreased the number of currently running processes by 1
- Released lock
- Checked if any process is currently running and if not then terminated the kernel
- Otherwise finished the user thread

Testing:

We checked if
- Exec
- Join
- Exit

Work on multiple processes and they worked.