OS Simulator Documentation

public final class **Kernel**	
private int clock	
public final int MAX_CLOCK	
public final SystemCalls systemCalls	
public final Scheduler scheduler	
public final Mutex mutex	
public final Memory memory	
public final Interpreter interpreter	
public final MemoryManagementUnit mmu	
public Kernel(int maxClock)	Initializes clock = 0, MAX_CLOCK = maxClock and all other attributes.
public void incrementClock()	Increments the clock, print the state of the OS and checks the Driver for new processes.
public int getClock()	
private void printState()	Prints out the current clock, memory content, the process IDs occupying memory blocks, the process IDs in all scheduler queues and the status of mutex locks.
public void run()	Starts the scheduler until simulation ends then closes the system call's scanner.
public void createProcess(String programName)	Creates a new PCB, uses systemCalls, interpreter and mmu to read the program from disk, parse its lines, allocate memory for it and save it to memory, then admits PCB to the scheduler. NOTE: If the mmu returns -1 addresses, call the mmu to swap the process to disk.
public void saveProcessState(Process process)	Saves the variable values from the process to their respective locations in the physical memory using systemCalls.
public Process restoreProcessState(PCB pcb)	Returns a process object populated with data read from the memory (using system calls) indexed by the pcb's memory table addresses. NOTE: If the memory table contains -1, call the mmu to swap the process from disk to memory.

public final class **SystemCalls**	
private final Kernel kernel	
public final Scanner scanner	Scanner object used for input, closed by Kernel at the end of the simulation.
public SystemCalls(Kernel kernel)	Initialize the scanner object.
public String[] readFromDisk(String fileName)	Returns an array of lines read from the file.
public void writeToDisk(String fileName, String lines[])	Writes lines to a file, creates the file if it was not already existing.
public String readFromScreen()	
public void writeToScreen(String content)	
public MemoryWord readFromMemory(int address)	
public void writeToMemory(int address, MemoryWord word)	

public final class **Variable**	
public final String name	
private String value	
public final int address	Logical address: index inside the PCB's memory table.
public Variable(String name, String value, int address)	
public String getValue()	
public void setValue(String value)	

public enum **ProcessState**
NEW, READY, RUNNING, WAITING, TERMINATED;

public final class **PCB**	
public final int pid	
private ProcessState state	
private int programCounter	
private final int[] memoryTable	
public PCB(int processSize)	Initializes pid using a unique generator. Initializes state to NEW and PC to 0.

	Initializes empty memoryTable array with the size.
public ProcessState getState()	
public void setState(ProcessState state)	
public int getPC()	
public void incrementPC()	
public int[] getMemoryTable()	Returns a **copy** of the array.
public void setMemoryTable(int[] table)	Assigns every element individually.

public final class **Process**	
public final PCB pcb	
private final String[] instructions	
private final HashMap<String, Variable> variables	Contains variable names and their variable objects for ease of access.
public Process(PCB pcb, String[] instructions, String[] varNames)	Initializes variables with empty value strings. Variables addresses start from instructions.length.
public Process(PCB pcb, String[] instructions, String[] varNames, String[] varValues)	
public String getInstruction(int index)	
public Variable getVariable(String name)	
public void setVariableValue(String name, String value)	
public Variable[] getAllVariables()	Returns an array of all the variables in the variables hashmap.

public final class **Interpreter**	
private final Kernel kernel	
public Interpreter(Kernel kernel)	
public String[] parseVariables(String[] programLines)	Returns an array of the variable names inside of the program.
public boolean executeInstruction(Process process)	Executes the instruction process.instructions[programCounter] using system calls. Increments the PC and returns true if execution was successful, otherwise tells the scheduler to block the process and returns false.

public final class **Memory**	
private final Kernel kernel	
public final int MAX_SIZE	

private final MemoryWord[] memory	
public Memory(Kernel kernel, int maxSize)	
public MemoryWord getMemoryWord(int address)	
public void setMemoryWord(int address, MemoryWord word)	Assigns the values inside the word to the MemoryWord at the indicated address.
public MemoryWord[] getMemory()	Returns a copy of the entire physical memory array.

public final class **MemoryWord**	
private String data	
private String varName	
private boolean isVariable	
private boolean isInstruction	
public MemoryWord()	Initializes empty strings and sets both flags to false.
public MemoryWord(String instruction)	Initializes data string, and empty varName and varVal strings, sets isInstruction to true and isVariable to false.
public MemoryWord(String varVal, String varName)	Initializes varName and data strings, sets isVariable to true and isInstruction to false.
public String getData()	
public void setData(String data)	
public String getVarName()	
public boolean isVariable()	
public boolean isInstruction()	
public void setWord(MemoryWord word)	Copies all attributes of the parameter word to the current word.
NOTE: We only need to use setData() in the case of changing a variable value, otherwise we create a new MemoryWord using the variable/instruction constructors and use setWord().	

public final class **MemoryManagementUnit**	
private final Kernel kernel	
private final boolean[] isAllocated	
private final int[] processIds	Contains IDs of PCBs associated with memory.

private final int[] tableOrder	Contains order of addresses in the PCB's memory table, useful for swapping to disk.
public MemoryManagementUnit(Kernel kernel)	Initializes the isAllocated, processIds and tableOrder arrays with the memory's max size, the first is initialized with false and the others with -1.
public int[] allocateMemory(int size, int pid)	Returns an array of unallocated addresses according to the indicated size, marks the addresses to true (allocated) and sets their processIds using pid. Also sets the order of the addresses returned in tableOrder. NOTE: If there isn't enough memory, return an array of full of -1.
public void deallocateMemory(int[] addresses)	Marks the addresses to false (unallocated) and sets their processIds to -1.
public void swapToDisk(Process process)	Save the process instructions and variables to a file named using its pid, ensure the instructions are saved according to their order from the tableOrder array. Ensure the PCB's memory table has -1 addresses, otherwise deallocate the memory.
public void swapFromDisk(PCB pcb)	Allocate memory for the process, if there isn't enough memory: swap other processes to disk until there is enough memory. Read the process instructions and variables from disk and save them to the allocated memory addresses.
public int[] getProcessIDs()	Returns a copy of the processIds array.

public final class **Scheduler**	
private final Kernel kernel	
public final int TIME_SLICE	
private final Queue<PCB> readyQueue	
private final Queue<PCB> inputWaitingQueue	
private final Queue<PCB> outputWaitingQueue	
private final HashMap<String, Queue<PCB>> fileWaitingQueue	

public Scheduler(Kernel kernel, int timeSlice)	Initializes kernel, TIME_SLICE and empty queues.
public void schedule()	Runs until the kernel clock reaches the MAX_CLOCK, each loop either dispatches a process or increments the kernel clock.
public void admitProcess(PCB pcb)	Changes a PCB's state to READY and adds it to the readyQueue.
private void dispatchProcess(PCB pcb)	Changes process state to RUNNING, dispatches 1st element in the readyQueue, restores process state using the kernel then executes the process using the interpreter.
public void blockProcess(Process process, String resource)	Changes process state to WAITING. Saves process state then adds the pcb to the waiting queue of the indicated resource (input, output, or file).
public void unblockProcess(String resource)	Removes the 1st PCB in the waiting queue and admits it to the ready queue.
private void interruptProcess(Process process)	Saves process state then changes it to READY and adds its PCB to the readyQueue.
private void terminateProcess(PCB pcb)	Changes process state to TERMINATED then deallocates its memory blocks.
public Queue<PCB> getReadyQueue()	Returns a copy of the queue.
public Queue<PCB> getInputWaitingQueue ()	
public Queue<PCB> getOutputWaitingQueue()	
public HashMap<String, Queue<PCB>> getFileWaitingQueue()	

public final class **Driver**	
private static Kernel kernel	
private static HashMap<Integer, ArrayList<String> processes	A list of processes that arrive at certain clock instances.
private Driver()	
public static void main(String[] args)	Enter program names and arrival times and initiliaze the kernel.
public static void checkProcessArrival(int clock)	Looks in the hashmap for processes that arrive at the indicated clock time.

public final class **Mutex**	
private final Kernel kernel	
private int inputSem	
private int outputSem	
private final HashMap<String, Integer> fileSem	
public Mutex(Kernel kernel)	Initializes all semaphores to 1.
public boolean semWait(String resource)	Decrements the semaphore and returns true if it was positive, otherwise returns false.
public void semSignal(String resource)	Increments the semaphore and tells the scheduler to unblock a waiting process.
public boolean isInputMutexFree()	Returns true if the semaphore is 1.
public boolean isOutputMutexFree()	
public String[] getMutexLockedFiles()	Returns an array of names of locked files.

Work Distribution

Mohamed Hossam	Abdelrahman Khalifa	Mohab Ashraf
• Scheduler • Mutex • Driver • Documentation • Testing	• Kernel • SystemCalls • Process • PCB • ProcessState • Variable	• Memory • MemoryWord • MMU • Interpreter

General Notes:

- Use only **if** guard clauses instead of **else** and **else if** blocks.
- Communicate between classes through the **Kernel** objects.
- Use **ArrayList<>** for dynamic lists, then use **.toArray(new String[0])** to convert them.
- Don’t change references of objects, change the values of the object’s attributes.
- Use **this.attribute** to refer to the class’s attribute.
- Always initialize attributes inside the constructor.
- Remember to catch index out of bound errors for array setters and getters.
- Primitive types get passed by value, non-primitive get passed by reference by value.
- You pass Wrapper Classes (ex: **Integer**) to **HashMap** declarations instead of primitives (ex: **Int**).