1. Logging (logger.c, logger.h)

Purpose: This module provides a centralized way to record events and execution details happening within the simulated operating system. This is crucial for debugging and understanding the system's behavior.

Key Components:

- **Function Pointers for GUI:**
  - add_event_gui_message: A function pointer intended to send event messages to a graphical user interface (GUI).
  - add_log_gui_message: A function pointer intended to send general log messages to a GUI.
  - These are NULL by default and need to be set by the GUI part of the application if it exists.
- **Logging Functions:**
  - log_execution(const char* format, ...):
    - This function is used to log general execution steps or debugging information.
    - It takes a format string and variable arguments (like printf).
    - If add_log_gui_message is set (i.e., the GUI is connected), it formats the message into a buffer and sends it to the GUI.
    - Otherwise, it prints the message to the console, prefixed with [DEBUG].
  - log_event(const char* format, ...):
    - This function is used to log significant events occurring in the system.
    - It also takes a format string and variable arguments.
    - If add_event_gui_message is set, it formats the message and sends it to the GUI.
    - Otherwise, it prints the message to the console, prefixed with [EVENT].
- **logger.h**:

- Declares the external function pointers and the two logging functions, making them available to other modules.
- Uses include guards (#ifndef LOGGER_H, #define LOGGER_H, #endif) to prevent multiple inclusions.

How it works: When other parts of the system (like memory management or mutex operations) perform an action or encounter an event, they call log_execution or log_event. These functions then decide whether to route the message to a GUI or print it to standard output.

2. Memory Management (memory.c, memory.h)

Purpose: This module simulates a simple computer memory, allowing for allocation, writing, and reading of data.

Key Components:

- **MemoryWord Structure:**
  - typedef struct { char content[100]; } MemoryWord;
  - Defines a single "word" of memory, which can hold a string of up to 99 characters plus a null terminator.
- **Global Memory Array:**
  - MemoryWord memory[MEMORY_SIZE];
  - A global array of MemoryWord structures, representing the total available memory. MEMORY_SIZE is defined as 60 in memory.h.
- **Functions:**
  - initializeMemory():
    - Iterates through the entire memory array.
    - Sets the content of each MemoryWord to "EMPTY", effectively clearing the memory.
  - allocateMemory(int size):
    - Implements a first-fit contiguous memory allocation strategy.
    - It searches for a block of size consecutive MemoryWords that are all marked "EMPTY".

        - If found, it returns the starting index (address) of this block.
        - If no such block is found, it returns -1.
    - writeToMemory(int address, const char* data):
        - Copies the given data string into the content field of the MemoryWord at the specified address. It uses strcpy, so ensure data is null-terminated and fits.
    - readFromMemory(int address):
        - Returns a pointer to the content (the string) stored at the specified address in memory.
    - storeVariableToMemory(const char* varName, const char* value):
        - Searches for the first "EMPTY" MemoryWord.
        - If found, it formats a string as "varName=value" and stores it in that memory location using snprintf for safety.
    - resetMemory():
        - Simply calls initializeMemory() to clear all memory contents.
- **memory.h**:
    - Defines MEMORY_SIZE and the MemoryWord structure.
    - Declares the global memory array and all the public memory management functions.

How it works: Processes in the simulated OS would use allocateMemory to request space, writeToMemory to store data, and readFromMemory to retrieve it. storeVariableToMemory provides a higher-level way to save variable-like data.

3. Mutex (Mutual Exclusion) (mutex.c, mutex.h)

Purpose: This module implements mutexes, which are synchronization primitives used to protect shared resources from concurrent access, preventing race conditions.

Key Components:

- **WaitNode Structure:**
    - typedef struct WaitNode { PCB* process; struct WaitNode* next; } WaitNode;

    - Represents a node in a linked list of processes waiting for a mutex. Each node holds a pointer to a PCB (Process Control Block) and a pointer to the next node.
- **Mutex Structure:**
    - typedef struct { int locked; int ownerPID; WaitNode* waitList; } Mutex;
    - locked: An integer (0 or 1) indicating if the mutex is free (0) or held (1).
    - ownerPID: The Process ID (PID) of the process currently holding the mutex; -1 if not held.
    - waitList: A pointer to the head of a linked list (WaitNode) of processes blocked on this mutex.
- **Global Mutex Variables:**
    - userInputMutex, userOutputMutex, fileMutex: Three global mutexes are declared to protect specific resources (user input, user output, and file access).
- **getResourceName(Mutex* mutex):**
    - A helper function that returns a string name for a given mutex, useful for logging.
- **Core Functions:**
    - semWait(Mutex* mutex, PCB* process): (Semaphore Wait, or P operation, Lock)
        - Logs an attempt to acquire the mutex.
        - **If the mutex is NOT locked (!mutex->locked)**:
            - The mutex is locked (mutex->locked = 1).
            - The ownerPID is set to the process->pid.
            - A log message confirms acquisition.
        - **If the mutex IS locked**:
            - The calling process is blocked. A log message indicates this.
            - The process state is updated to BLOCKED (using updatePCBState, presumably from pcb.c).

        - A new WaitNode is created for the process and added to the end of the mutex->waitList.
  - semSignal(Mutex* mutex): (Semaphore Signal, or V operation, Unlock)
    - Logs the release attempt, showing the owner if there is one.
    - **If the waitList is empty**:
      - The mutex is simply unlocked (mutex->locked = 0).
      - ownerPID is set to -1.
    - **If the waitList is NOT empty**:
      - It searches the waitList to find the process with the **highest priority** (lowest mlfqLevel). This is important for MLFQ scheduling.
      - This highest-priority process (unblocked) is removed from the waitList.
      - The unblocked process is chosen to acquire the mutex next:
        - The mutex locked status remains 1 (or is set to 1).
        - ownerPID is set to unblocked->pid.
        - A log message confirms the unblocked process acquired the mutex.
      - **Scheduling Adjustments for the Unblocked Process:**
        - The programCounter of the unblocked process is incremented (to move past the semWait instruction).
        - Its state is updated to READY.
        - **If currentAlgorithm == MLFQ**:
          - mlfqTicksUsed is set to the full quantum of its current mlfqLevel (implying it used its turn trying to acquire the lock).
          - If it was at mlfqLevel == 1, it gets demoted to mlfqLevel = 2.

                - It's then enqueued into the appropriate MLFQ queue.
            - **Else (FCFS or RR)**:
                - It's enqueued into the general readyQueue.

- **mutex.h**:
    - Defines the WaitNode and Mutex structures.
    - Declares the global mutex variables and the semWait and semSignal functions.

How it works: When a process needs access to a shared resource (e.g., console input), it calls semWait on the corresponding mutex. If the resource is busy, the process blocks and is added to a queue. When the process holding the mutex releases it using semSignal, the highest priority waiting process is awakened and granted access. The MLFQ-specific logic in semSignal ensures fairness and priority adjustments.

4. PCB (Process Control Block) (pcb.c, pcb.h - pcb.h not fully shown but inferred)

Purpose: The PCB is a data structure that stores all the information the operating system needs about a particular process.

Key Components (from pcb.c and common PCB knowledge):

- **PCB Structure (inferred from pcb.c and pcb.h):**
    - pid: Process ID, a unique identifier.
    - state: Current state of the process (e.g., NEW, READY, BLOCKED, RUNNING, TERMINATED). ProcessState is an enum type.
    - priority: Process priority (though MLFQ levels might be more actively used).
    - programCounter: Stores the address of the next instruction to be executed for this process.
    - lowerBound, upperBound: Define the memory address space allocated to this process.
    - quantumUsed: For Round Robin, tracks how much of its current time slice the process has used.
    - mlfqTicksUsed: For MLFQ, tracks ticks used within the current quantum at its current level.

- mlfqLevel: The current queue level in a Multi-Level Feedback Queue (MLFQ) scheduler (initialized to 1, highest priority).
- variables[26][0]: An array likely to store single-character variable names (A-Z) and their values, though only the first char [0] is shown being set to \0. The actual storage mechanism isn't fully detailed here.
- justUnblocked: A flag, possibly to handle specific scheduling decisions immediately after unblocking.
- waitTime: Total time spent in the READY state.
- blockTime: Total time spent in the BLOCKED state.
- lastStateChangeTime: The value of clockCycle (a global variable) when the process last changed its state. Used to calculate waitTime and blockTime.

- **Functions:**
  - createPCB(int pid, int lower, int upper):
    - Allocates and initializes a new PCB structure.
    - Sets the pid, programCounter (to lower), memory bounds (lower, upper).
    - Initializes state to NEW.
    - Sets default priority, MLFQ level to 1, and other counters/timers to zero or initial values.
    - Initializes the variables array (clears variable names).
    - Sets lastStateChangeTime to the current clockCycle.
  - updatePCBState(PCB* pcb, ProcessState state):
    - This function is called when a process changes its state (e.g., from READY to RUNNING, or RUNNING to BLOCKED).
    - It calculates the time spent in the *previous* state by subtracting pcb->lastStateChangeTime from the current global clockCycle.
    - If the previous state was READY, this duration is added to pcb->waitTime.

        - If the previous state was BLOCKED, this duration is added to pcb->blockTime.
        - Finally, it updates pcb->state to the new state and resets pcb->lastStateChangeTime to the current clockCycle.

How it works: Every process in the system will have an associated PCB. The scheduler uses information in the PCBs to decide which process to run next. When a process's status changes (e.g., it blocks on I/O, or its time slice expires), its PCB is updated accordingly. The updatePCBState function is crucial for tracking how long processes spend waiting or blocked, which is useful for performance analysis.

Interactions and System View:

- **Processes (PCBs) and Memory:** When a process is created (createPCB), it's assigned memory bounds (lowerBound, upperBound). It would then use functions from memory.c to interact with its allocated memory space.
- **Processes (PCBs) and Mutexes:** When a process needs a shared resource, it calls semWait with its PCB. If it blocks, its PCB's state is set to BLOCKED, and it's added to the mutex's waitList. semSignal will change a waiting PCB's state back to READY and re-integrate it into the scheduler's queues.
- **Scheduler (External, but referenced):**
    - The code references mlfq[5] (an array of queues for MLFQ), blockedQueue, readyQueue, and currentAlgorithm. This implies a scheduler module exists that manages these queues and selects processes to run based on the currentAlgorithm.
    - The mutex.c file directly enqueues unblocked processes into mlfq or readyQueue.
- **Logging:** All modules (memory, mutex, pcb, and presumably the scheduler and instruction execution loop) would use logger.c to record their actions and important state changes. For example, mutex.c logs extensively when processes attempt to wait, acquire, or release mutexes, and when they are blocked or unblocked. pcb.c also has a commented-out log line in updatePCBState.