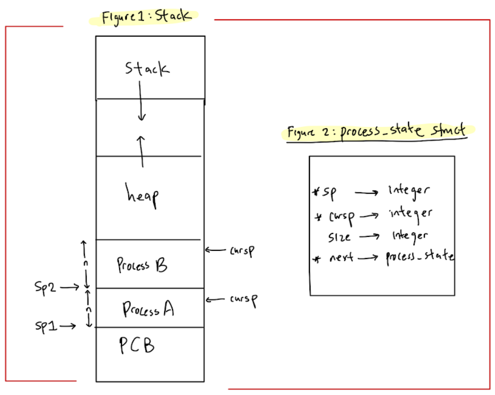
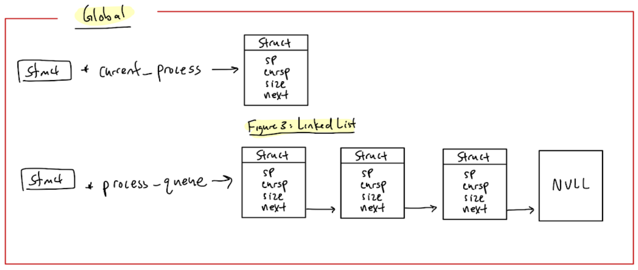
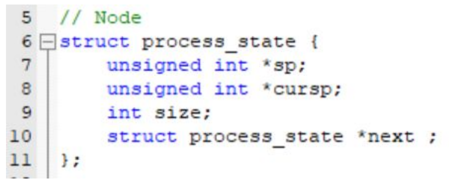
Eldor Bekpulatov (eb654)
Carol Zhang (cz233)

Lab 3: Concurrency



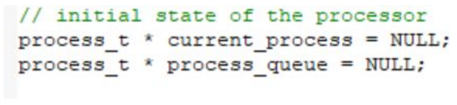


a) design and coding



```
// Node
struct process_state {
    unsigned int *sp;
    unsigned int *cursp;
    int size;
    struct process_state *next ;
};
```

process_state is a structure that helps us interface with individual processes by modeling them as nodes. It is designed to store information such as a pointer to the beginning of a heap memory block containing values related to a specific process. In simpler terms, it contains the address to a memory location where the actual values related to a process are stored. It encapsulates the size of the memory block for in field called **size** and appropriate stack pointer called **cursp** which points to the top of the stack within that allocated memory block for that process. The last field of this structure is a pointer to another instance of process_state called **next**. This allows us to model the list of ready-to-run processes as a linked list data structure.

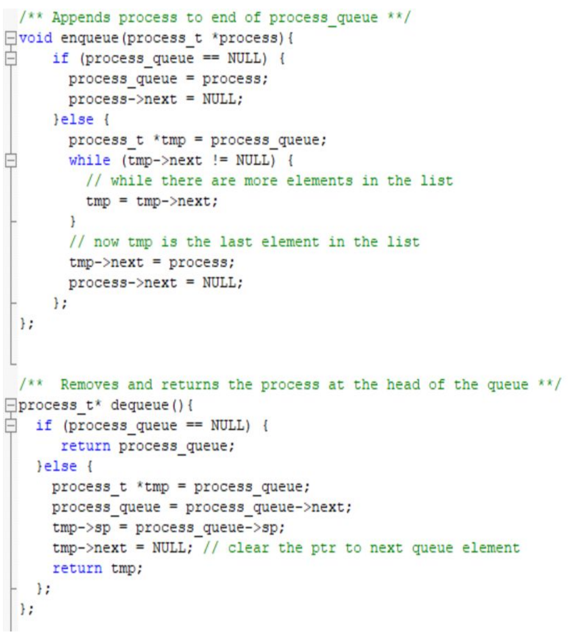


```
// initial state of the processor
process_t * current_process = NULL;
process_t * process_queue = NULL;
```

In order to deal with multiple processes we needed a way to keep track of processes and assure fair CPU time is guaranteed for each. To keep track of the unfinished processes we implemented queue of **process_state**. This was easily implemented using linked list structure. We initialized our global variables **process_queue** as **NULL** by default, since no process will be waiting to execute in the beginning of the program. We also set the **current_process** to **NULL**, since no process will be executing in the beginning.

To keep our logic less convoluted, we designed our **process_queue** and **current_process** to be mutually exclusive. We didn't want our current process to be linked in any way to our queue of ready-to-run processes. At no point in execution of our program, our current process will be pointing at any of the nodes from process queue, or vice versa. This design choice ensures that our queue will never become a circular linked-list.

In order to manage queue of ready-to-run processes, we defined some helper functions that made adding and removing nodes easier and followed queueing conventions. For example, **enqueue(*p)** links the process to the last linked node of the queue. The **deque()**, on the other hand, unlinks the first node in the beginning of the list and returns that node with all outgoing links removed.



```
/** Appends process to end of process_queue **/
void enqueue(process_t *process){
    if (process_queue == NULL) {
      process_queue = process;
      process->next = NULL;
    }else {
      process_t *tmp = process_queue;
      while (tmp->next != NULL) {
        // while there are more elements in the list
        tmp = tmp->next;
      }
      // now tmp is the last element in the list
      tmp->next = process;
      process->next = NULL;
    };
};

/**  Removes and returns the process at the head of the queue **/
process_t* dequeue(){
  if (process_queue == NULL) {
     return process_queue;
  }else {
    process_t *tmp = process_queue;
    process_queue = process_queue->next;
    tmp->sp = process_queue->sp;
    tmp->next = NULL; // clear the ptr to next queue element
    return tmp;
  };
};
```

Figure 4. Helper Functions

process_create

The first function we implemented, process_create, creates a process that starts at function f with an initial stack size of n. We use the function process_stack_init in 3140_concurr.c to allocate space in memory for the new process. This function will use **malloc()** to create **n** slots for stored content in the heap which then returns the stack pointer pointing to the beginning of the newly created process. If allocation was unsuccessful, the given function will return **NULL**, then our function returns -1 to indicate un-successful creation of process. If allocation was successful, the stack is initialized to all zeros and a pointer to the beginning of the memory block is returned. If memory was correctly allocated, then we start modeling this process as a node under structure we have diagrammed in Figure 2 above. This structure's **sp** and **cursp** fields will be pointing to the beginning of the memory block that we received in the last step. This is because our process hasn't started yet so our newly allocated memory block is empty. The **size** field stores the size of the allocated memory block in the heap. The "next" field is set to **NULL**, because this process will be queued to the end of our list. Once, our node is ready to be queued, we must allocate space for it in heap, and we do that using **malloc()**. This is visualized in Figure 5. Lastly, we enqueue the process to the **process_queue** and return 0 to indicate successful creation of a process.

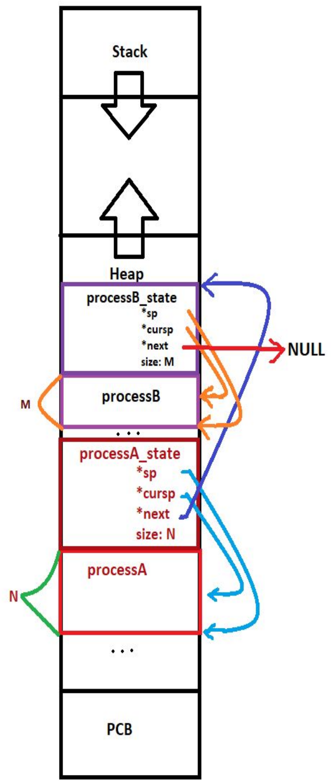


Figure 5. Memory Allocation Scheme

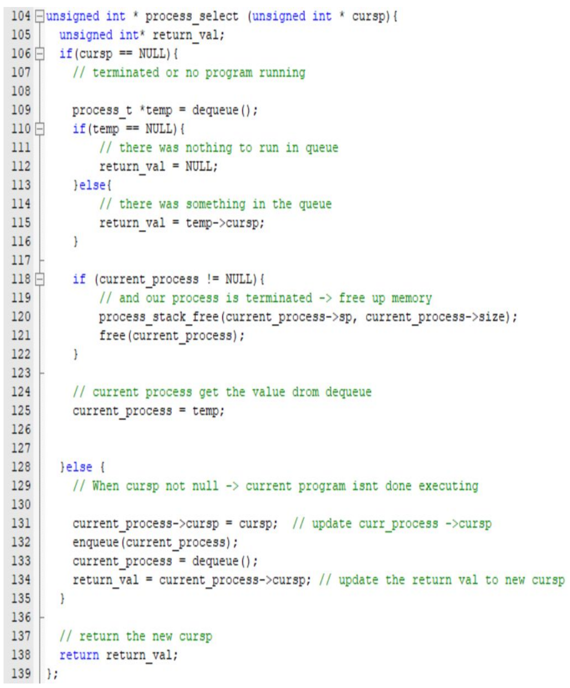
process_start

The next implemented function, **process_start**, initializes an external periodic interrupt timer and sets off the processor to start executing form process queue. Steps to set up the interrupt timer include: enabling PIT interrupt handler, enabling clock to the PIT module, and loading a value to the timer. This timer is loaded with a value which represents the processor's atomic time/maximum allowed time allowed to be spent on a single process. We assigned the value 0x3D090 (which corresponds to a 250,000 clock cycles). We kept the timer disabled, since the provided function **process_begin()** is configured to kick off the processor and enable the timer. The final step of the function calls **process_begin()**.

process_select

The final function, **process_select** returns the value of the next stack pointer for the next ready process in the queue. For instance, if we have finished the current process in the last atomic period, we must now decide if we need to re-enqueue it or move on to the next element in **process_queue** to be executed in the next atomic period. This function is called by the provided PIT0 interrupt handler code which was configured in process_start.

The design of our code starts by checking if the **cursp** is **NULL** (line 106), meaning the process has either terminated or the processor has never begun. By dequeuing, we check if there's anything anything to run. If the **process_queue** was empty, **temp** is will be **NULL**, so we set our **return_val** to **NULL** (line 112). If there was something in the **process_queue**, we set out **return_val** to the newly dequeued process's **cursp** (line 115). By checking if **current_process** is **null**, we decide if we have to deallocate memory (lines 118-122). Now that we covered the case **cursp** is **NULL**, we need to cover the cases if the **cursp** is not **NULL**, meaning the currently running process has not terminated (lines 128-134). If the process had not finished, we update **current_process->cursp** (line 131), and enqueue the **current_process**. Now, that we have saved the last state of our current process, we need to dequeue a new process state from **process_queue** (line 133) and return it's **cursp** (line 134).



```
104 unsigned int * process_select (unsigned int * cursp){
105   unsigned int* return_val;
106   if(cursp == NULL){
107     // terminated or no program running
108
109     process_t *temp = dequeue();
110     if(temp == NULL){
111         // there was nothing to run in queue
112         return_val = NULL;
113     }else{
114         // there was something in the queue
115         return_val = temp->cursp;
116     }
117
118     if (current_process != NULL){
119         // and our process is terminated -> free up memory
120         process_stack_free(current_process->sp, current_process->size);
121         free(current_process);
122     }
123
124     // current process get the value drom dequeue
125     current_process = temp;
126
127
128   }else {
129     // When cursp not null -> current program isnt done executing
130
131     current_process->cursp = cursp;  // update curr_process ->cursp
132     enqueue(current_process);
133     current_process = dequeue();
134     return_val = current_process->cursp; // update the return val to new cursp
135   }
136
137   // return the new cursp
138   return return_val;
139 };
```

b) code review

For code review, we used the debugger to see where errors were thrown. This consisted mostly of using the debugger to check the PIT register fields constantly. It was especially helpful to see whether certain registers were loaded or cleared when we wanted them to be. Another instance of code review was during pair programming, if someone saw that there was an error or warning, the other person would be notified.

c) testing

Once we had evidence of our program running, we first played with the PIT0's LDVAL values. We started off with extremely large LDVAL (~90 Million cycles). This was to show that if the atomic time was large enough each process was running independently as if one process finished executing before the beginning of the second process. Resulting pattern was 5 blinks of red then 5 blinks of blue, then green led would turn on. To check if our concurrency algorithm was actually working, we lowered the LDVAL to 250K cycles. This was good enough to see the evidence of two processes running concurrently. This time, both red and blue leds were blinking seemingly instantaneously. Once we passed our provided benchmark. We implemented additional test cases to cover different scenarios.

Test case 1: when we just have one program running. This was useful to see if our program can handle running one process and switching back into the same process in the next atomic period of the CPU. We made red led toggle 5 times in one process. Our observed behavior was that, red led toggled 5 times and green led turned on as expected.

Test case 2: when we have three processes running and they one by one each process starts terminating while leaving the others to continue. This was useful to see if our concurrency algorithm can scale up to more than 2 processes and handles asynchronous programs. Since we had access to 3 led's we settled to test only three processes at once. If our algorithm can handle 3 independent processes, it should be able to handle more processes the same way. So, designed the test to blink red leds independently 5 times, blue leds 4 times and green leds 3 times. Finally, turn green led on at the end and keep it on forever. Our observed behavior was (r-g-b) white led blinks 3 times, (r-b) purple blinks once, and red led blinks once. Then, green turn on and stays on forever.

d) documenting/writing

Documenting the lab was done through Google Docs, where each member practiced live editing and communicated through tools such as text comments. Any updates were recorded, reviewed and annotated by each member. This method of documentation proved to be highly effective.

e) work distribution

In order to collaborate, we met up three times during the week to work together and get the general expectations of the lab. We practiced peer programming, where each member took the lead when necessary. Eldor worked on process_select() and figured out test cases. Carol and Eldor worked together on process_start(), process_create() and other helper structures and functions. Debugging and stress testing was done by Eldor, while documenting and providing visuals were done by Carol. Both members worked on lab report. There were many issues initially in getting our code to work, but once we figured out our bugs, coming up with tests were pretty simple.