CS 111 Discussion 1A/D

WEEK 4

Brian Roysar

Logistics

- Lab 1 (Pipe Up) due tonight
- Lab 2 (Round Robin) will be released today, will be due in two weeks time

Lab 1 Q&A

Scheduling

- What is scheduling?
  - Computers run more than one task “simultaneously”
  - Efficiently allocate CPU resources among these various tasks/processes
- Key metrics
  - Fairness
  - Waiting/Response time
    - **Avg Waiting Time:** The time a process spends idle in the queue
    - **Avg Response Time:** The time between process arrives and first time it is scheduled
  - CPU Utilization
  - Throughput

Common approaches to scheduling

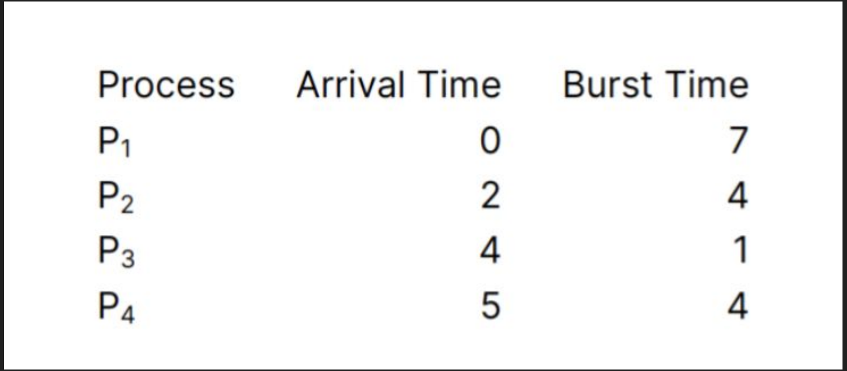
1. **First come first serve**
    a. The first process that arrives gets the CPU resources. First in, first out
2. **Shortest job first**
    a. Schedule the process with the shortest burst time first
3. **Shortest remaining time first**
    a. Same as SJF, but preemptive
4. **Round Robin**
    a. Divide time into time slices, and cycle between queued processes depending on the set quantum length

First come first serve

The first process that arrives gets the CPU resources.

- First in, first out
  - Implemented with a queue

- How will the OS assign these processes to the CPU? What is the average waiting time for all processes?



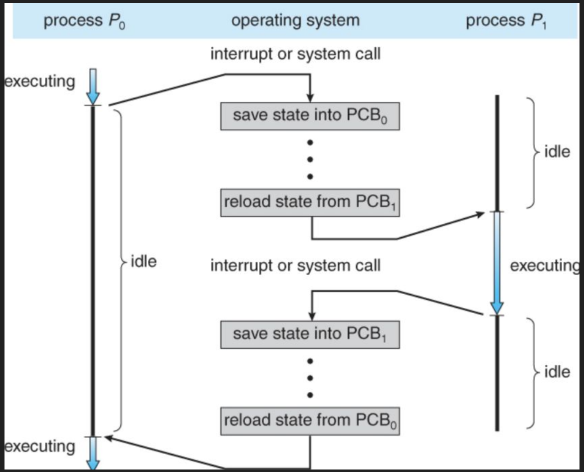
Process	Arrival Time	Burst Time
P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4

Pros and Cons to FCFS

Advantages	**Disadvantages**
Easy to implement	Increased waiting time (Large process comes first, and a lot of small processes come after)
Minimize context switching, since we are only switching processes when we are family done with one	Does not work well with time sharing systems
	Unfair approach, non-preemptive

Aside: Context Switching

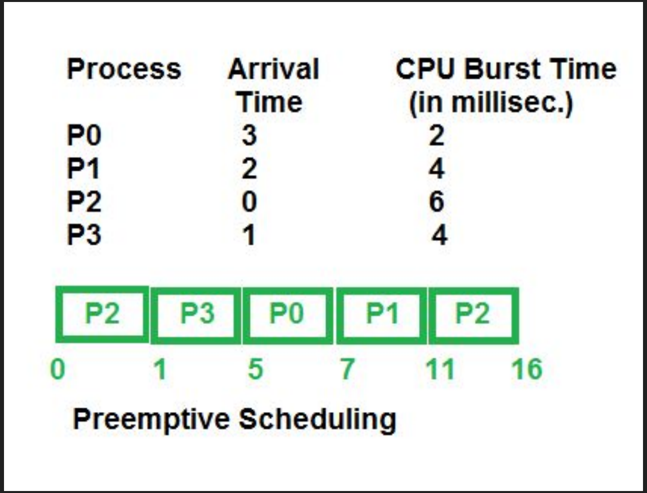
- Involves the following two things
  - Saving the current process context/information so that it can be used later
  - Loading the execution context of another process to allow it to run.



This operation is expensive as it takes resources and time to perform

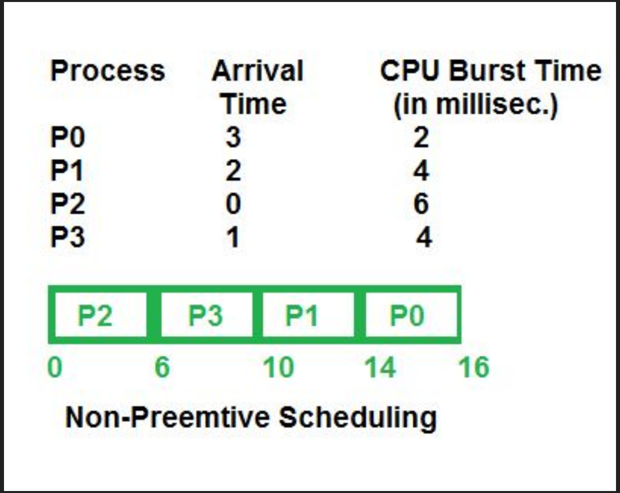
Thus, when choosing a scheduling algorithm, we want to minimize the number of context switches we do

Aside: Preemptive vs. Non preemptive



Process	Arrival Time	CPU Burst Time (in millisec.)
P0	3	2
P1	2	4
P2	0	6
P3	1	4

Preemptive Scheduling

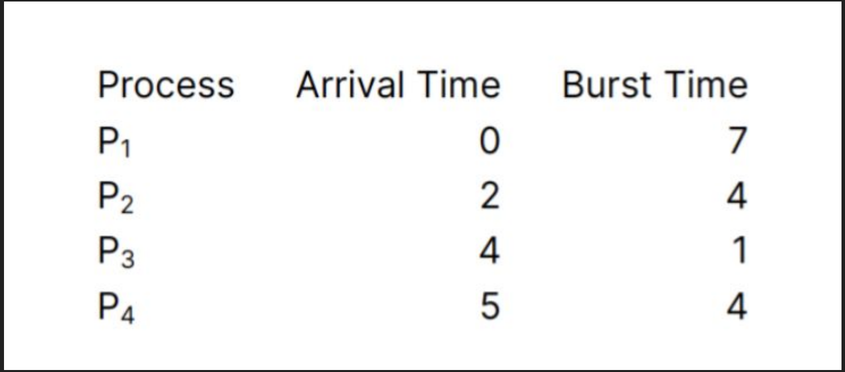


Process	Arrival Time	CPU Burst Time (in millisec.)
P0	3	2
P1	2	4
P2	0	6
P3	1	4

Non-Preemtive Scheduling

Shortest Job First and Shortest Remaining Time First

- SJF: Schedule the process with the shortest burst time first
  - **Non preemptive**
- SRTF: Same as SJF, **but preemptive**
- How will the OS assign these processes to the CPU? What is the average waiting time for all processes? How many context switches?



Process	Arrival Time	Burst Time
P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4

Pros and Cons to SJF

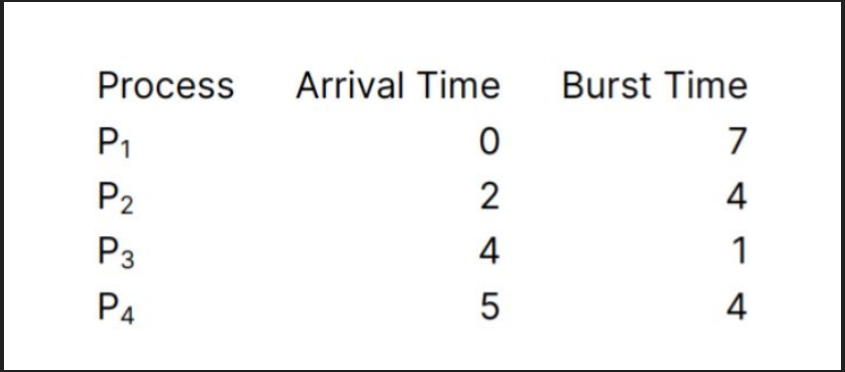
Advantages	**Disadvantages**
Increased throughput since we are prioritising shortest jobs	Impossible to predict the future
Minimize average waiting time	Starvation for long processes
Minimize context switching, since we are only switching processes when we are family done with one	

Pros and Cons to SRTF

Advantages	**Disadvantages**
Shortest jobs favored – increased throughput	Impossible to predict the future
Minimize average waiting time	Starvation for long processes
	More context switches than SJF. Preemptive

Round Robin

- We have seen scheduling algorithms that prioritizes processes based on their time. But, how do we keep this **fair?** Round robin aims to keep it fair.
  - Divide time into time slices, and cycle between queued processes depending on the set quantum length
- How will the OS assign these processes to the CPU? What is the average waiting time for all processes? Assume QS = 3 How many context switches?



Process	Arrival Time	Burst Time
P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4

Pros and Cons to Round Robin

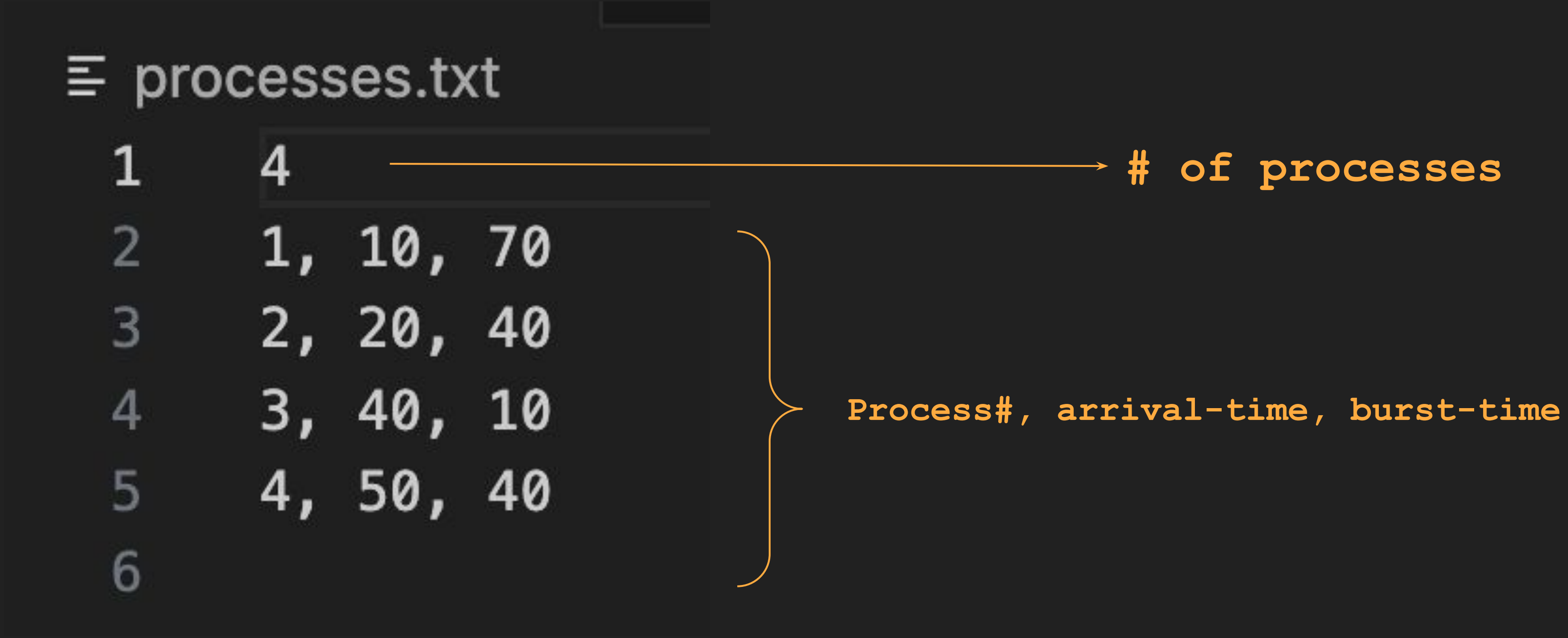
Advantages	**Disadvantages**
Fair/Minimize starvation. All processes regardless of arrival time/burst time will get CPU soon	Performance depends on quantum size ● Low means we are doing a lot of context switches ● High means we are approaching FCFS
Low average response time	No sense of priority between tasks. Too fair?

Lab 2

Project 2

- Implementing round robin scheduling with a given scheme for quantum length and list of processes and related info
- Quantum length
    - **<u>Fixed size</u>** - `./rr processes.txt 30`
        - *“Simulate round robin on processes.txt with quantum length 30”*
- Assume that we only have a single CPU
- Output
    - Average waiting time
    - Average response time

The input: processes.txt



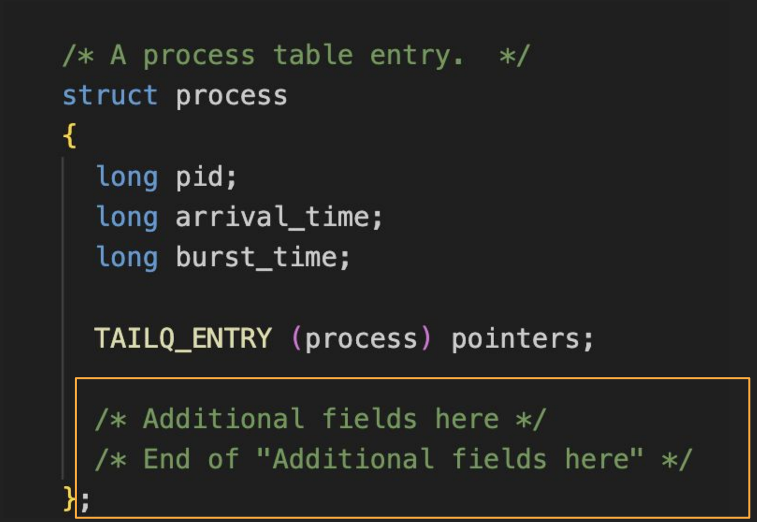
processes.txt

```
4
1, 10, 70
2, 20, 40
3, 40, 10
4, 50, 40

```

Very similar to the tables we were previously working on!

rr.c

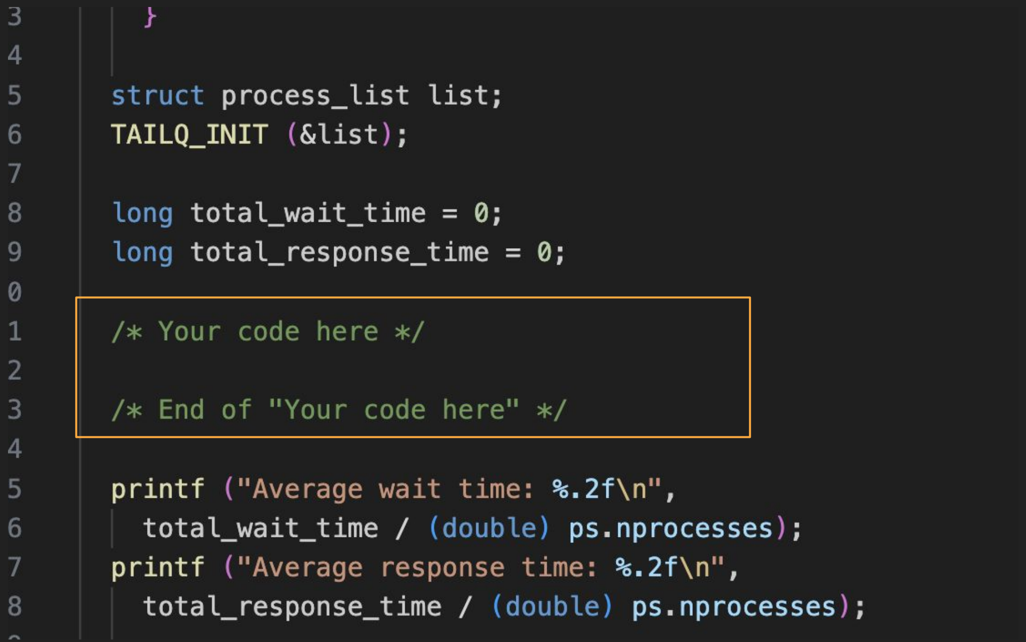


```c
/* A process table entry.  */
struct process
{
  long pid;
  long arrival_time;
  long burst_time;

  TAILQ_ENTRY (process) pointers;

  /* Additional fields here */
  /* End of "Additional fields here" */
};
```

Add additional fields that will help you perform calculations



```c
    }

  struct process_list list;
  TAILQ_INIT (&list);

  long total_wait_time = 0;
  long total_response_time = 0;

  /* Your code here */

  /* End of "Your code here" */

  printf ("Average wait time: %.2f\n",
    total_wait_time / (double) ps.nprocesses);
  printf ("Average response time: %.2f\n",
    total_response_time / (double) ps.nprocesses);
```

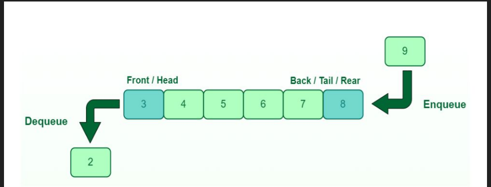
Simulate Round Robin while updating the fields in the process struct

Round Robin

As a scheduling algorithm, Round Robin maintains to be fair by keeping a queue of waiting processes and schedules them in a First-in-First-out manner (FIFO)

What data structure helps us simulate this FIFO behavior?

- A queue!
- Using a queue, we can build our process queue, and have processes that arrive first processed first, as well as maintaining order between other incoming processes



Queue Data Structure

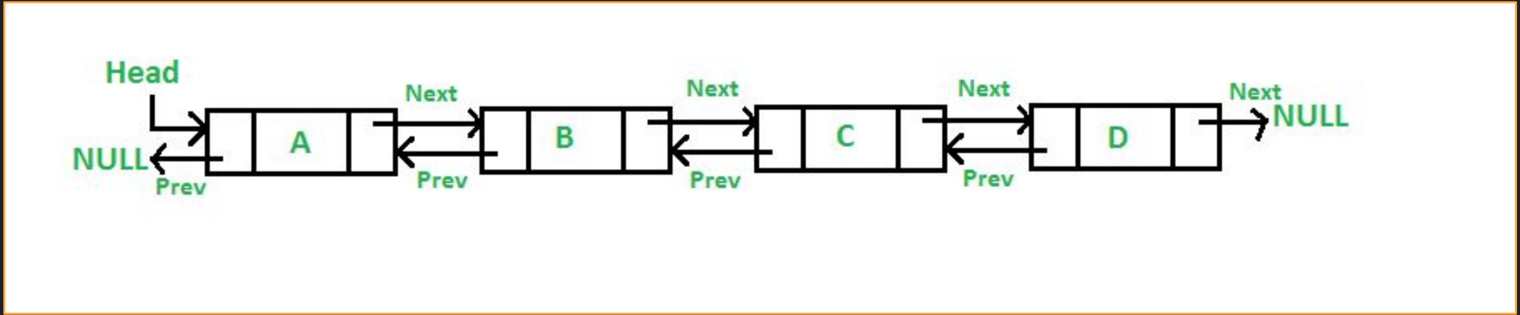
How can we implement queues in C?

Linked Lists in C

TAILQ from sys/queue.h

Doubly linked list

In this project, we will be using sys/queue.h to help us simulate the FIFO behavior that RR requires



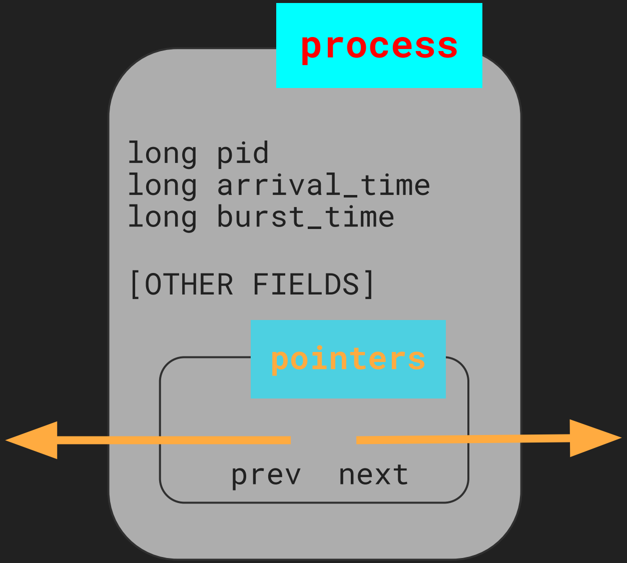
- Each node has two pointers
  - **Prev:** pointing to the node before
  - **Next:** pointing to the node after
- Allows for bidirectional travel

Representing Nodes

```
/* A process table entry.  */
struct process
{
  long pid;
  long arrival_time;
  long burst_time;

  TAILQ_ENTRY (process) pointers;

  /* Additional fields here */
  /* End of "Additional fields here" */
};
```

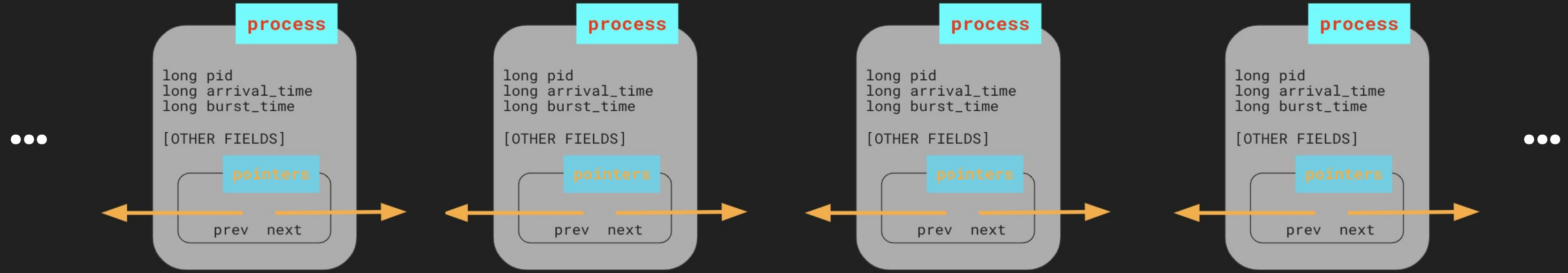


TAILQ_ENTRY(TYPE) pointers;

Macro that allows us to create a pair of pointers to previous and next nodes of type process

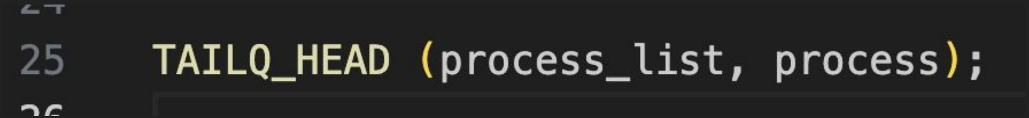
```
#define TAILQ_ENTRY(type)
struct {
        struct type *tqe_next;  /* next element */
        struct type **tqe_prev; /* address of previous next element */
}
```

Process Nodes in the Bigger Picture



Initializing the linked list

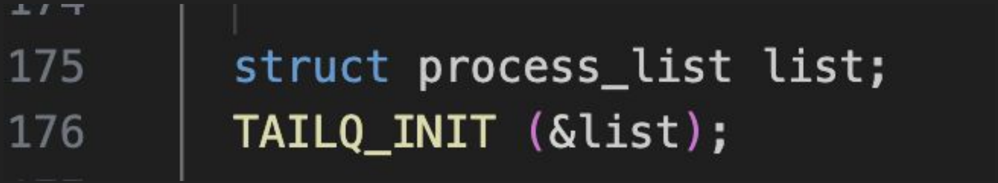
We've set up the building blocks to represent our linked list, lets initialize it



```
TAILQ_HEAD (process_list, process);
```

Create a struct that will act as our head of our linked list.

- Pass in node type, which in this case is process
- Head struct will be called process_list



```
    struct process_list list;
    TAILQ_INIT (&list);
```

Initializes the queue and allows us to use the head, list, as an abstraction of our entire linked list.

Appending/Deleting

To insert elements into the linked list, we use `TAILQ_INSERT_*` macros:

To insert to the front of the linked list:

```
TAILQ_INSERT_HEAD(head_pointer, pointer_to_new_element, links)
```

To insert the new element at the end of the linked list:

```
TAILQ_INSERT_TAIL(head_pointer, pointer_to_new_element, links)
```

To insert the new element after the element prev_node:

```
TAILQ_INSERT_AFTER(head_pointer,pointer_to_prev_element,pointer_to_new_element,li
nks)
```

To **remove** an element from the linked list:

```
TAILQ_REMOVE(head_pointer, pointer_to_element_to_remove, links)
```

Traversal

To traverse the linked list, here are a few important macros to use:

TAILQ_FOREACH(): traverses the list in the forward direction

TAILQ_NEXT(): returns the next element on the list, NULL if current is the last element

TAILQ_FIRST(): returns the first element on the list

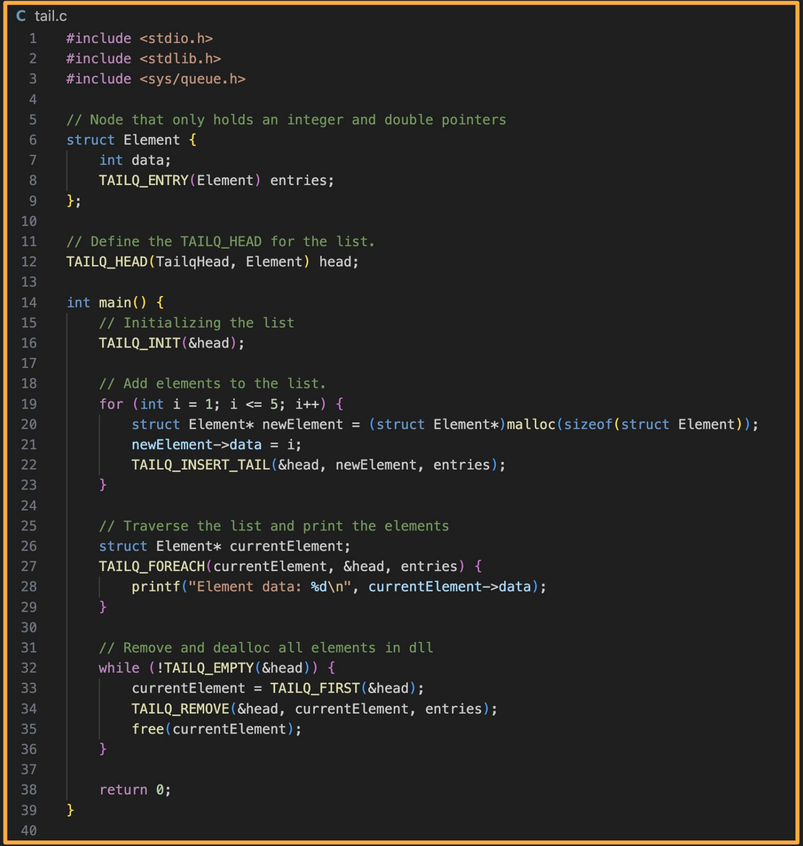
TAILQ_LAST(): returns the last element on the list

TAILQ_PREV(): returns the previous element on the list

TAILQ_FOREACH_REVERSE():traverses the list in the reverse order

For more details about each macro, check out
https://man7.org/linux/man-pages/man3/tailq.3.html

Example on how to use TAILQ



tail.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/queue.h>

// Node that only holds an integer and double pointers
struct Element {
    int data;
    TAILQ_ENTRY(Element) entries;
};

// Define the TAILQ_HEAD for the list.
TAILQ_HEAD(TailqHead, Element) head;

int main() {
    // Initializing the list
    TAILQ_INIT(&head);

    // Add elements to the list.
    for (int i = 1; i <= 5; i++) {
        struct Element* newElement = (struct Element*)malloc(sizeof(struct Element));
        newElement->data = i;
        TAILQ_INSERT_TAIL(&head, newElement, entries);
    }

    // Traverse the list and print the elements
    struct Element* currentElement;
    TAILQ_FOREACH(currentElement, &head, entries) {
        printf("Element data: %d\n", currentElement->data);
    }

    // Remove and dealloc all elements in dll
    while (!TAILQ_EMPTY(&head)) {
        currentElement = TAILQ_FIRST(&head);
        TAILQ_REMOVE(&head, currentElement, entries);
        free(currentElement);
    }

    return 0;
}

```

Exercise:

1. How would you take the first element in queue, print out its contents, and then put it back into the back of the queue?

```
1 <-> 2 <-> 3 <-> 4 <-> 5

        Output: 1

2 <-> 3 <-> 4 <-> 5 <-> 1
```

Project Milestones

1. Familiarize yourself with what is given
   a. Process text file, starter code
2. Set up and get used to using TAILQ doubly linked list
3. Manually run through examples to see what output should be with given test case
4. Figure out how to have a clock/time that we will be basing scheduling on
   a. Hint: while loop
5. Figure out what additional fields we must store in the node structure
6. Implement the static quantum size scheduler

References

https://media.geeksforgeeks.org/wp-content/cdn-uploads/gq/2014/03/DLL1.png
https://linux.die.net/man/3/tailq_entry
https://github.com/openbsd/src/blob/master/sys/sys/queue.h
Past TA slides (Can Aygun)