CS111 Winter 2024

Discussion 1C/F (week4)

Yadi Cao

Logistics

- Lab 1 is due this Fri (Apr/27)
- Last Dis: **Hints** for Lab1
- This Dis: Lab 2 background knowledge
- Next Fri Dis (Apr/03): **Lab 02 Code skeleton**

Agenda

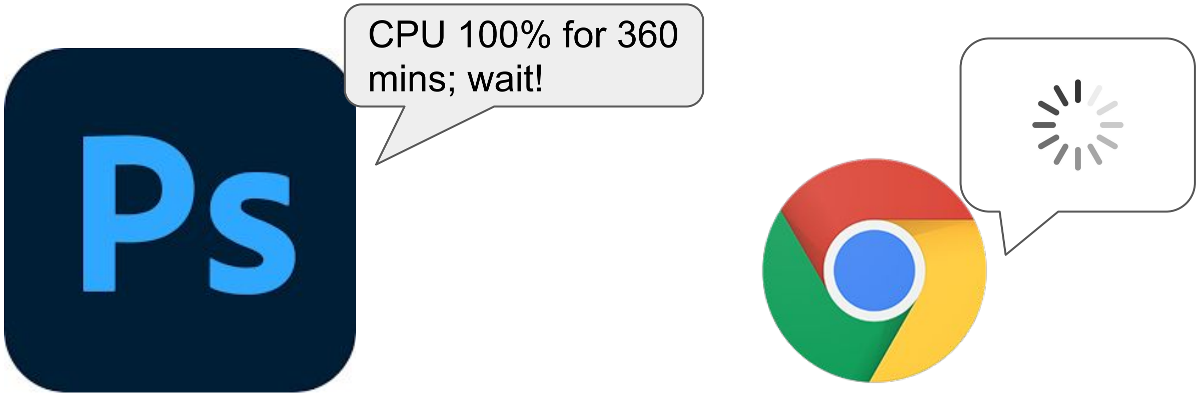
- What is scheduling, why do we need a good scheduling?
- Typical scheduling policies, PROs and CONs.
- Lab2 background (Round-robin, concrete example)
- Lab2 starter (coding)

Lab2 Background

Why do we need a scheduling?

Can you depict a modern computer without scheduling?

- You are opening a browser to get some picture resources, but the background PhotoShop is rendering forever…you have to wait for PS to finish



- <u>Formal ans:</u> because CPU is a **preemptable resources**; sharing CPU time between processes can increase **efficiency and fairness**; crucial for **concurrency**

Why do we need a good scheduling?

- The scheduler obeys the strict ordering of incoming tasks

CPU 100% for 360 mins; others wait

I said wait!

Hey; we just need a sec to shoot a message

First Come First Served (FCFS)

Problems?

- so even tiny tasks like email takes 0.5s to send, if they arrive late, they have to wait
- <u>Formal ans:</u> **wait and response** time will be too long

Why do we need a good scheduling?

- The scheduler now is smarter, it always picks up the easiest job 1st

Lighters 1st thie time!

Not bad just wait a while

Lighters 1st!!!
out-of-the-way!!!

Shortest Job First (SJF)

Problems?

- The heavy job never gets even started
- <u>Formal ans:</u> **starves** heavy jobs

Shortest Job First (SJF), continued

- Always schedule the job with the shortest burst time first
- No-preemption (don't interrupt the running process)

<u>Provable, this is optimal at minimizing average wait time</u>

Besides **starving**, any other practical problems?

- CPU doesn't know the burst time before running a task (estimate)

Shortest Remaining Time First (SRTF)

- Improved SJF with preemption:
  - When a new task added, compare its burst time with current task remaining time.
  - If the new task has smaller burst time, swap.

Same problems as SJF!

- CPU doesn't know the remaining time until a task finishes (then remaining time is zero)
- Starving the heavy jobs

Metrics

- Minimize waiting time and response time
  Don’t have a process waiting too long (or too long to start)

- Maximize CPU utilization
  Don’t have the CPU idle

- Maximize throughput
  Complete as many processes as possible

- Fairness
  Try to give each process the same percentage of the CPU

What is Lab 2 about?

- **Implement round-robin (RR) scheduling**
  - You need to write **"a simulation"** for round-robin (RR) scheduling for the specified workload and the quantum length.
  - You'll be given a basic skeleton that parses an input file and command line arguments.
  - To be able to do this lab successfully, you'll need to understand how round-robin scheduling works and utilize a data structures(which?) needed.
  - The output is average waiting and response time for this **"simulation run"**

- **Example output**

```
cs111@cs111 cs111/lab2-pan (main *%) » make
cc -std=gnu17 -Wpedantic -Wall -O0 -pipe -fno-plt -fPIC   -c -o rr.o rr.c
cc -lrt -Wl,-O1,--sort-common,--as-needed,-z,relro,-z,now  rr.o   -o rr
cs111@cs111 cs111/lab2-pan (main *%) » ./rr processes.txt 3
Average waiting time: 7.00
Average response time: 2.75
```

UCLA dining hall is Round Robin

Me,
Hungry
FIFO

Quantum(limited amount) of food

FIFO

Quantum of food

FIFO

Quantum of food

FIFO

Done!

Round Robin

- **Round Robin** is a scheduling algorithm that optimizes fairness and response time
  - The operating system divides execution into time slices (or quanta), and an individual time slice is called a quantum
  - It maintains a FIFO (First in First Out) queue of processes
  - A process will be preempted if it is still running at end of quantum and re-add to the end of the queue

- **Round Robin** is similar to **Dining hall in Google**
  - Everyone first comes to the hall in a FIFO queue
  - The staff will let you sequentially pick up a quantum types of food (also limited amount)
  - At the end of the quantum, either you are done, satisfied with the food and leave
  - If not, say you need more shrimp, rejoin the end of queue (also hide your plate), and pick up a second round

Round Robin Example (in exam 2 years ago)

Assume RR with a quantum length of 3 units, and the 4 tasks arrive/run(burst) time are listed as below

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

- Q1: Draw the trajectory of occupancy(which task) of CPU
- Q2: Draw the trajectory of the Queue of the unfinished tasks
- Q3: Calculate the average wait/response time

Definitions

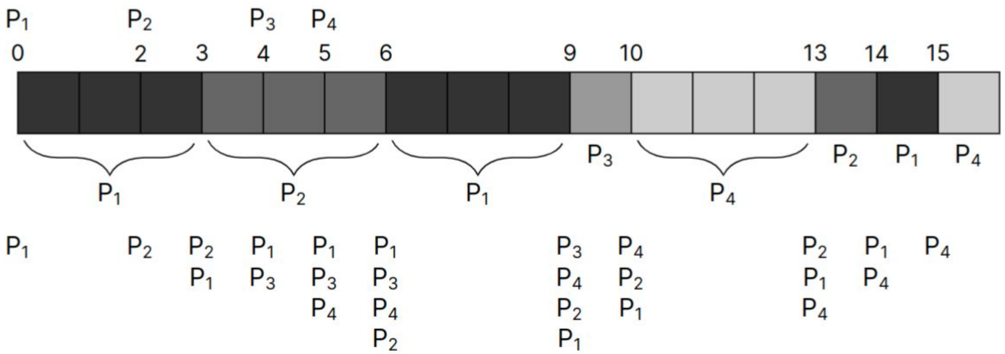
```
waiting_time = end_time - arrival_time - burst_time

response_time = start_execute_time – arrival_time
```

Draw animation and replace the below

RR with a quantum length of 3 units

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

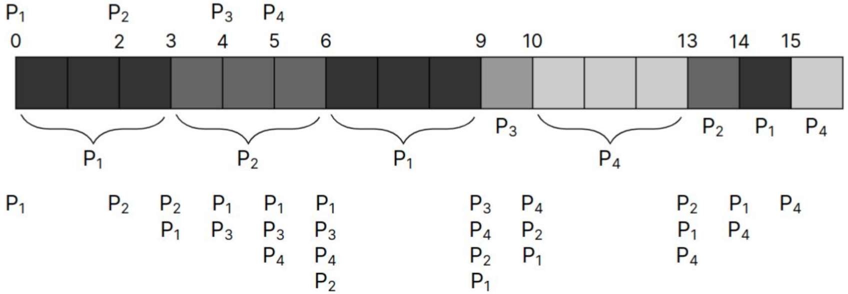
For RR, our schedule is (arrival on top, queue on bottom):



Round Robin Example

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

For RR, our schedule is (arrival on top, queue on bottom):



T=0, P1 arrives, add P1 the front of the queue
T=2, P2 comes in, added to the queue

T=3, P1 is done, P2 is schedule 3 units, P1 is on top now
T=4, P3 comes in, sent to the back of the queue
T=5, P4 comes in, sent to the back of the queue

T=6, P2 is done, added to bottom, P1 is on top and runs

T=9, P1 is done, P3 is on top and runs

T=10, P3 is done, P4 is on top and runs

T=13, P4 goes to the back, P2 is on top and runs

T=14, P1 is on top and runs with 1 unit

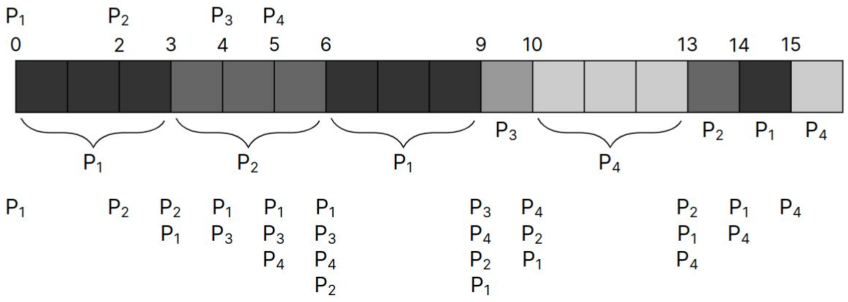
T=15, P4 is left and runs with 1 unit

Note: processes with same arrival time should be added to queue in order

Round Robin Example

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

For RR, our schedule is (arrival on top, queue on bottom):



Average waiting time

The elapsed time a process is waiting for other processes since its arrival

P1:

- finishes at 15, starts at 0 and it takes 7
- 15-0-7=8

P2:

- finishes at 14, starts at 2 and it takes 4
- 14-2-4=8

P3:

- finishes at 10, starts at 4 and it takes 1
- 10-4-1=5

P4:

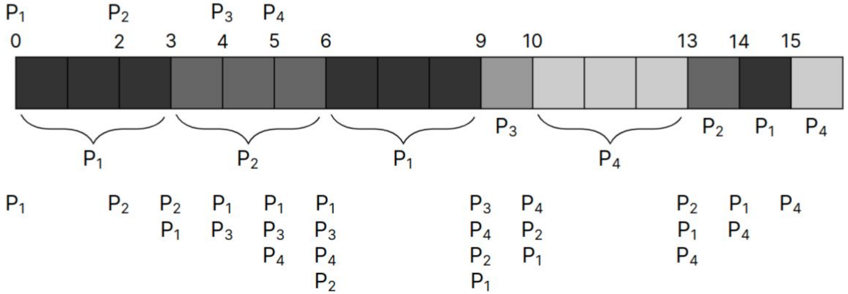
- finishes at 16, starts at 5 and it takes 4
- 16-5-4=7

Average waiting time: $\frac{8+8+5+7}{4} = 7$

Round Robin Example

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

For RR, our schedule is (arrival on top, queue on bottom):



Average response time

The time starting from its first arrival to when it first starts executing

P1:

- It starts at 0, arrives at 0
- 0

P2:

- It starts 3, arrives at 2
- 3-2=1

P3:

- It starts at 9, arrives at 4
- 9-4=5

P4:

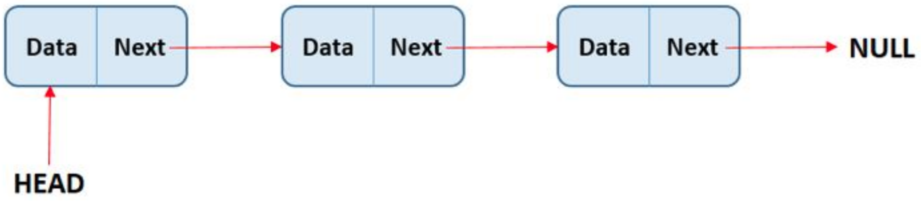
- It starts at 10, arrives at 5
- 10-5=5

Average response time: $\frac{0+1+5+5}{4} = 2.75$

Now we want to implement, which data structure?

FIFO queue by linked list

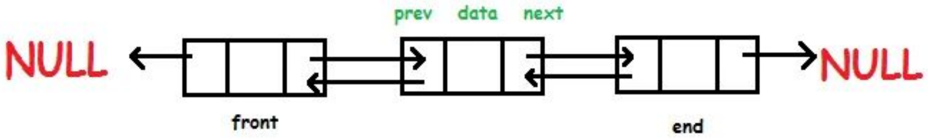
- A linked list is a **linear dynamic** data structure to store data items.
- Unlike arrays, the linked list does not store data items in contiguous memory locations
- Instead, the elements in a linked list are "**linked**" together using **pointers**.



Doubly Linked List

What is a doubly linked list (DLL)

- A type of linked list in which each node contains data as well as two pointers.
- One to the next, another to the previous container
- The head node's previous ptr points to NULL
- The tail node's next ptr points to NULL



The two pointers help us to traverse the list in both backward and forward direction.

Great but we don't have linked list in C

We are writing kernel code, we use C.

C doesn’t have a linked list type, unlike C++. What are your options ?

Option 1: Implement it ourselves

- We might need to define a node struct (i.e, the element), then lookup, delete, insert, length methods to access them
- Of course, this requires a lot of “fun” pointer operations

Option 2: “steal” something already existed

- We just need to import it and use it !
- But in order to use it correctly, we need to understand it correctly!

`tailq`

See the appendix :>

Don't Reinvent The Wheel, Unless You Plan on Learning More About Wheels

https://blog.codinghorror.com/dont-reinvent-the-wheel-unless-you-plan-on-learning-more-about-wheels/

Lab 2 Skeleton

Lab02 Overview

- Implement part of a scheduling **“simulation”**
  - Schedule processes based on Round-Robin algorithm
  - Implement scheduling queue using doubly-linked list (TAILQ macros)
  - Calculate average waiting and response times
- Example Output

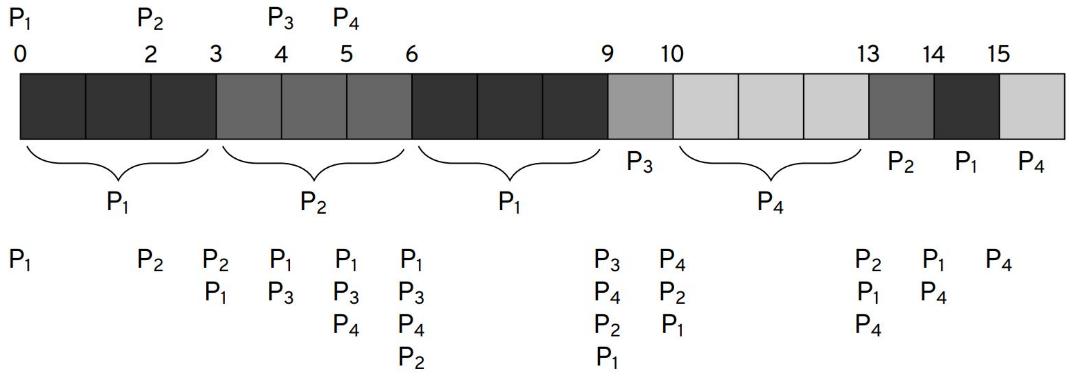
```
cs111@cs111 cs111/lab2-pan (main *%) » make
cc -std=gnu17 -Wpedantic -Wall -O0 -pipe -fno-plt -fPIC   -c -o rr.o rr.c
cc -lrt -Wl,-O1,--sort-common,--as-needed,-z,relro,-z,now  rr.o   -o rr
cs111@cs111 cs111/lab2-pan (main *%) » ./rr processes.txt 3
Average waiting time: 7.00
Average response time: 2.75
```

Process.txt

```
4
1, 0, 7
2, 2, 4
3, 4, 1
4, 5, 4
```

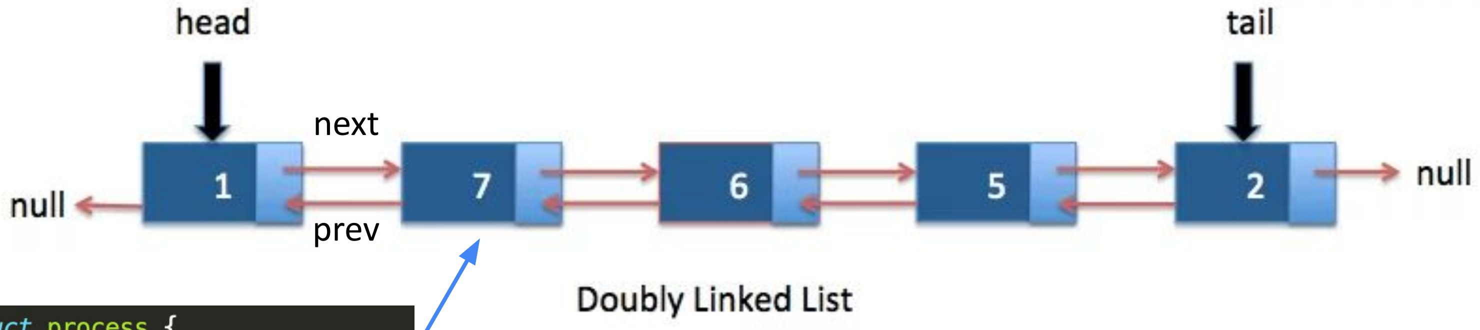
Process	Arrival Time	Burst Time
P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4

For RR, our schedule is (arrival on top, queue on bottom):



Doubly Linked List

`struct process {}` declares a process node structure on a doubly linked list (queue)




```c
struct process {
  u32 pid;
  u32 arrival_time;
  u32 burst_time;

  u32 remaining_time;
  u32 start_exec_time;
  u32 waiting_time;
  u32 response_time;

  TAILQ_ENTRY(process) pointers;

};
```

Skeleton code

```c
#include <errno.h>
#include <fcntl.h>
#include <stdbool.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/queue.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <unistd.h>

typedef uint32_t u32;
typedef int32_t i32;

struct process {
  u32 pid;
  u32 arrival_time;
  u32 burst_time;

  TAILQ_ENTRY(process) pointers;

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

TAILQ_HEAD(process_list, process);
```

```c
int main(int argc, char *argv[])
{
  if (argc != 3) {
    return EINVAL;
  }
  struct process *data;
  u32 size;
  init_processes(argv[1], &data, &size);

  u32 quantum_length = next_int_from_c_str(argv[2]);

  struct process_list list;
  TAILQ_INIT(&list);

  u32 total_waiting_time = 0;
  u32 total_response_time = 0;

  /* Your code here */
  /* End of "Your code here" */

  printf("Average waiting time: %.2f\n", (float) total_waiting_time / (float) size);
  printf("Average response time: %.2f\n", (float) total_response_time / (float) size);

  free(data);
  return 0;
}
```

./rr processes.txt 3

`&data` is the array of all processes parsed from txt

`size` is the number of processes from txt

```
init_processes(argv[1], &data, &size);
```

TXT

processes.txt

Arrival time will not always in order

quantum_time

Should be >0

Invalid arg exit with EINVAL (optional)

While(!finished)

```
struct process *data;
```

All processes parsed from txt

P1 P2 P3 P4

Check arrival time (t)

Add newly arrived process to queue

If multiple process, insert based on top to bottom order in txt

```
struct process_list list;
```

`list` is the scheduling queue of active processes

P1 P2

Queue can be empty

After newly arrived process

pop the 1st process in the queue

Remove process from head of queue

(pop may have done the removal)

If (remaining time > 0){

insert unfinished process to end of queue

}

P1

- Run process for 1 unit at a time for a total of min(quantum length, remaining time)
- Decrease process remaining time
- Check loop termination condition
- ++t

```
If all
data[i].remaining
time == 0
```

Break while; finished

Calculate avg wait time & response time

Creating the queue

`TAILQ_ENTRY(NODE);`

- Define **entries to the next/ previous** `NODE` in the queue
- `NODE` is the user defined node structure

`TAILQ_HEAD(CONTAINER, NODE);`

- Define **a *container*** that stores **all** the `NODE`s in the queue
- `CONTAINER` is a newly defined ***structure*** (e.g. use *struct* `CONTAINER` node_list; to initialize a `CONTAINER` instance node_list)
- Always followed by `TAILQ_INIT`(&node_list); this when the initialization of the queue occurs

```
struct process {
  u32 pid;
  u32 arrival_time;
  u32 burst_time;

  TAILQ_ENTRY(process) pointers; //macro for creating pointer

};

TAILQ_HEAD(process_list, process);
```

```
int main(int argc, char *argv[])
{
  //...

  //list (queue) of active processes
  struct process_list list;
  TAILQ_INIT(&list);
```

Modifying the queue

```
struct process {
    u32 pid;
    u32 arrival_time;
    u32 burst_time;

    TAILQ_ENTRY(process) pointers; //macro for creating pointer

};

TAILQ_HEAD(process_list, process);
```

```
int main(int argc, char *argv[])
{
    //...

    //list (queue) of active processes
    struct process_list list;
    TAILQ_INIT(&list);
```

- `void TAILQ_INSERT_TAIL(TAILQ_HEAD *head, struct NODE *elm, TAILQ_ENTRY ENTRY);`

```
struct process *p;
p = &data[i];

TAILQ_INSERT_TAIL(&list, p, pointers);
```

- `TAILQ_FOREACH(struct NODE *var, TAILQ_HEAD *head, TAILQ_ENTRY ENTRY);`

```
TAILQ_FOREACH(p, &list, pointers){
```

- `void TAILQ_REMOVE(TAILQ_HEAD *head, struct NODE *elm, TAILQ_ENTRY ENTRY);`

```
TAILQ_REMOVE(&list, p, pointers);
```

Acknowledgement

1. Previous TAs: Lu Pan, Tianxiang Li, Salekh Parkhati, and Alex Tiard
2. Jonathan Eylofson , UCLA Computer Science, Fall21, cs111 course slides
3. https://www.softwaretestinghelp.com/linked-list/
4. https://www.studytonight.com/data-structures/doubly-linked-list
5. https://stackoverflow.com/questions/22315213/minimal-example-of-tailq-usage-out-of-sys-queue-h-library/22319023
6. https://man7.org/linux/man-pages/man3/TAILQ_NEXT.3.html
7. https://nxmnpg.lemoda.net/3/TAILQ_FIRST

Thanks!

Appendix
APIs for linked list in C

`tailq` API

- An abstract **two-way operation of a queue**
- `tailq` (tail queue) is defined in the `<sys/queue.h>` file
  - There are other data structures (such as singly linked list) too in `<sys/queue.h>`, but we will use the tail queue
- A `tailq` supports the following functionality:
  - **Insertion** of a new entry at the head of the list
  - **Insertion** of a new entry before or after any element in the lis
  - **Removal** of any entry in the list
  - **Forward traversal** through the list
  - **Reverse traversal** through the list

https://github.com/freebsd/freebsd-src/blob/master/sys/sys/queue.h#L308

`tailq` API

There is a bunch of macros defined in <sys/queue.h>

And they define and operate on doubly linked tail queues

Manuals:

- https://man7.org/linux/man-pages/man3/TAILQ_NEXT.3.html
- https://nxmnpg.lemoda.net/3/TAILQ_FIRST

TAILQ_ENTRY - Define a node

First, you need to define what elements do you want to store in the queue (or for simplicity, in the doubly linked list, in our case)

```
TAILQ_ENTRY(TYPE);
```

- **Declare a pointer structure** that connects the elements in the queue.
- `TYPE` is **name of user defined structure**, which contains field **`TAILQ_ENTRY`**, named **NAME**
- **Establish pointers** used to insert items into list. You include it into your structure (node) you want to list up

```
// define a node structure in the queue
struct my_node {
      int num;
      // this is the data part

      TAILQ_ENTRY(my_node) pointers;
      // this part corresponds to the pointer part
}
```


```
struct process {
  u32 pid;
  u32 arrival_time;
  u32 burst_time;

  TAILQ_ENTRY(process) pointers;
};
```

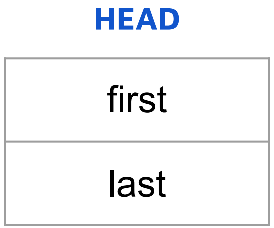
TAILQ_HEAD - Define a head

A tail queue is headed by a structure defined by the **TAIL_HEAD()** macro. In our case, this is **the head of the linked list**. This structure abstracts the whole linked list we are creating.

```
TAILQ_HEAD(HEADNAME, TYPE) head;
```

- **Define a head structure** acting as *container* for list elements
- **HEADNAME** is the name of a user defined structure
- struct **TYPE** is the **type of element (node)** to be linked into the queue (linked list)
- Structure created by the **TAILQ_HEAD()** **contains a pair of pointers**, one to the first element in the queue and the other to the last element in the queue.

```
struct my_node {
        int num;
        TAILQ_ENTRY(my_node) pointers;
}
TAILQ_HEAD(my_linked_list, my_node);

// now we can use the newly defined struct:
// declare LL of type my_linked_list
struct my_linked_list LL;
```




```
struct process {
  u32 pid;
  u32 arrival_time;
  u32 burst_time;

  TAILQ_ENTRY(process) pointers;
};

TAILQ_HEAD(process_list, process);
```

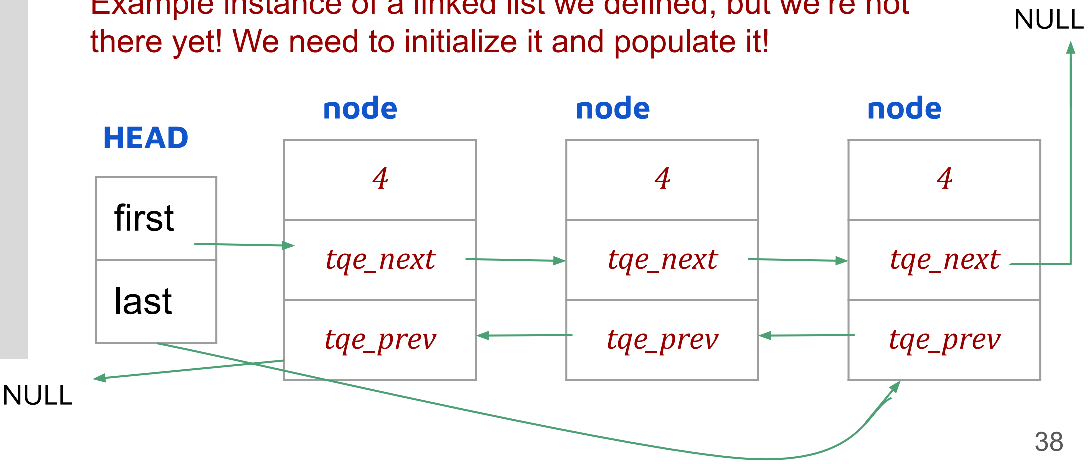
TAILQ_HEAD - Define a head

```
TAILQ_HEAD(HEADNAME, TYPE);
```

- **Define a head structure** acting as *container* for list elements
- **HEADNAME** is the name of a user defined structure
- struct **TYPE** is the **type of element (node)** to be linked into the queue (linked list)
- Structure created by the `TAILQ_HEAD()` **contains a pair of pointers**, one to the first element in the queue and the other to the last element in the queue.

```
struct my_node {
      int num;
      TAILQ_ENTRY(my_node) pointers;
}
TAILQ_HEAD(my_linked_list, my_node);

// now we can use the newly defined struct:
// declare LL of type my_linked_list
struct my_linked_list LL;
```

Example instance of a linked list we defined, but we're not there yet! We need to initialize it and populate it!



`TAILQ_INIT` - Initialize a queue

So far, we defined the necessary building blocks of our linked list, i.e, the node type and the head. In order to use it, we need to initialize one:

```
void TAILQ_INIT(TAILQ_HEAD *head);
```

- **Initializes the queue**
- **head** is the **pointer to a head element of type** we just defined
- The head element kind of abstracts the whole linked list

```
struct my_node {
      int num;
      TAILQ_ENTRY(my_node) pointers;
}
TAILQ_HEAD(my_linked_list, my_node);

//now we can use the newly defined struct:
struct my_linked_list LL;

// We initialize a new linked list
TAILQ_INIT(&LL);
```

Now, we have initialized a linked list of `my_node` elements (although it's still empty now), we can have all types of linked list fun!

TAILQ_INSERT_* - Insert into the linked list

```
TAILQ_INSERT_HEAD(head_pointer, pointer_to_new_element, pointers);
```

- To insert the new element to the front of the linked list

```
void TAILQ_INSERT_TAIL(TAILQ_HEAD *head, struct TYPE *elm, TAILQ_ENTRY NAME);
TAILQ_INSERT_TAIL(head_pointer, pointer_to_new_element, pointers);
```

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

```
TAILQ_INSERT_BEFORE(pointer_to_next_element, pointer_to_new_element, pointers);
```

- To insert the new element before the element next_node

```
TAILQ_INSERT_AFTER(head_pointer, pointer_to_prev_element, pointer_to_new_element, pointers);
```

- To insert the new element after the element prev_node

TAILQ_INSERT_* - Insert into the linked list

```
void TAILQ_INSERT_TAIL(TAILQ_HEAD *head, struct TYPE *elm, TAILQ_ENTRY NAME);
TAILQ_INSERT_TAIL(head_pointer, pointer_to_new_element, pointers);
```

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

```
struct my_node {
        int num;
        TAILQ_ENTRY(my_node) pointers; // define a pointer structure: my_node
}
TAILQ_HEAD(my_linked_list, my_node); // define a head structure: my_linked_list

struct my_linked_list LL; // declare LL of type my_linked_list
TAILQ_INIT(&LL); // initialize a new linked list LL

struct my_node *new_node = malloc(sizeof(struct my_node));
TAILQ_INSERT_HEAD(&LL, new_node, pointers); // insert the node: new_node
```

Note: Here **pointers** refer to the struct name which contains the prev and next pointers

TAILQ_REMOVE - Remove from the linked list

```
void TAILQ_REMOVE(TAILQ_HEAD *head, struct TYPE *elm, TAILQ_ENTRY NAME);
TAILQ_REMOVE(head_pointer, pointer_to_element_to_remove, pointers);
```

- To remove an element from the linked list

```
struct my_node {
        int num;
        TAILQ_ENTRY(my_node) pointers;
}
TAILQ_HEAD(my_linked_list, my_node);

struct my_linked_list LL;
TAILQ_INIT(&LL);

//insert a new node: new_node1
struct my_node * new_node1 = malloc(sizeof(struct my_node));
TAILQ_INSERT_HEAD(&LL, new_node1, pointers);

//insert a new node: new_node2
struct my_node * new_node2 = malloc(sizeof(struct my_node));
TAILQ_INSERT_TAIL(&LL, new_node2, pointers);

//delete anode: new_node2
TAILQ_REMOVE(&LL, new_node2, pointers);
free(new_node2); // free the memory
```

TAILQ_FOREACH - Traverse a queue

```
TAILQ_FOREACH(struct TYPE *var, TAILQ_HEAD *head, TAILQ_ENTRY NAME);
TAILQ_FOREACH(current_element, head_pointer, pointers);
```

- Traverse the queue referenced by **head** in the forward direction

```
struct my_node {
        int num;
        TAILQ_ENTRY(my_node) pointers;
}
TAILQ_HEAD(my_linked_list, my_node);
struct my_linked_list LL;
TAILQ_INIT(&LL);

struct my_node *current_node;
struct my_node *new_node1 = malloc(sizeof(struct my_node)); //insert a new node: new_node1
TAILQ_INSERT_HEAD(&LL, new_node1, pointers);

struct my_node *new_node2 = malloc(sizeof(struct my_node)); //insert a new node: new_node2
TAILQ_INSERT_TAIL(&LL, new_node2, pointers);

TAILQ_FOREACH(current_node, &LL, pointers) {
      // do something
}
```

https://man7.org/linux/man-pages/man3/tailq.3.html

TAILQ_FIRST - Index the first item on the queue

```
TAILQ_FIRST(TAILQ_HEAD *head);
```

- Returns the first item on the queue, or NULL if the queue is empty

```
struct my_node {
        int num;
        TAILQ_ENTRY(my_node) pointers;
}
TAILQ_HEAD(my_linked_list, my_node);

struct my_linked_list LL, L0;
TAILQ_INIT(&LL);

//insert a new node: new_node1
struct my_node * new_node1 = malloc(sizeof(struct my_node));
TAILQ_INSERT_HEAD(&LL, new_node1, pointers);

//insert a new node: new_node2
struct my_node * new_node2 = malloc(sizeof(struct my_node));
TAILQ_INSERT_TAIL(&LL, new_node2, pointers);

//index the first item
L0 = TAILQ_FIRST(&LL);
```

TAILQ_EMPTY - Check for an empty queue

```
TAILQ_EMPTY(TAILQ_HEAD *head);
```

- Evaluates to true if there are no items on the tail queue

```
struct my_node {
        int num;
        TAILQ_ENTRY(my_node) pointers;
}
TAILQ_HEAD(my_linked_list, my_node);

struct my_linked_list LL, current;
TAILQ_INIT(&LL);

struct my_node * new_node1 = malloc(sizeof(struct my_node));
TAILQ_INSERT_HEAD(&LL, new_node1, pointers);

struct my_node * new_node2 = malloc(sizeof(struct my_node));
TAILQ_INSERT_TAIL(&LL, new_node2, pointers);

While (!TAILQ_EMPTY(&LL)) {
       current = TAILQ_FIRST(&list);
      // do something
}
```

A minimal example

```
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/queue.h> // header file

// define a node structure: entry
struct entry {
    int data;
    TAILQ_ENTRY(entry) entries; /* Tail queue */
};

// define a head node structure: tailhead
TAILQ_HEAD(tailhead, entry);
```

tailq is very powerful!

https://man7.org/linux/man-pages/man3/tailq.3.html

```
int main(void)
{
    struct entry *n1, *n2, *n3, *np;        /* Tail queue node */
    struct tailhead head;                   /* Tail queue head */

    TAILQ_INIT(&head);                      /* Initialize the queue */

    n1 = malloc(sizeof(struct entry));
    TAILQ_INSERT_HEAD(&head, n1, entries);  /* Insert n1 at the head */

    n1 = malloc(sizeof(struct entry));
    TAILQ_INSERT_TAIL(&head, n1, entries);  /* Insert n1 at the tail */

    n2 = malloc(sizeof(struct entry));
    TAILQ_INSERT_AFTER(&head, n1, n2, entries); /* Insert n2 after n1 */

    n3 = malloc(sizeof(struct entry));
    TAILQ_INSERT_BEFORE(n2, n3, entries);       /* Insert n3 before n2 */

    /* Forward traversal */
    int i = 0;
    TAILQ_FOREACH(np, &head, entries) {
        np->data = i++; // access member: data of a structure
    }

    /* TailQ deletion */
    while (!TAILQ_EMPTY(&head)) {
        n1 = TAILQ_FIRST(&head);
        TAILQ_REMOVE(&head, n1, entries);
        free(n1);
    }
}
```