CS 537
Section 4: Data Structures in C

Michael Swift

Linux Completely Fair Scheduler

- Written in 42 hours by Ingo Molnar at RedHat
- No scheduling queues
  - Runnable processes stored in a red-black tree
    - Left = run sooner, right = run later
  - Processes ordered by future time when they should run
- Order tasks by "wait_runtime" field – indicating the greatest need for the CPU
  - When one task runs, the other tasks that are waiting for the CPU are at a disadvantage:
    - the current task gets an unfair amount of CPU time.
    - this fairness imbalance is expressed and tracked via wait_runtime.
    - "wait_runtime" is the amount of time the task should now run on the CPU for it to become completely fair and balanced.
- For example:
  - If there are 3 tasks running, after that runs, second is behind by equal amount of time.

More CFS

- Fairness
  - Idea: because you waited longer, you deserve more CPU later
  - rungueue fair_clock value tracks the 'CPU time a runnable task would have fairly gotten, had it been runnable during that time'.
    - As a task gets into the rungueue, the current time is recorded
    - while the process waits for the CPU, its wait_runtime value gets incremented by an amount depending on the number of processes currently in the rungueue.
  - rg->fair_clock values measure the 'expected CPU time' a task should have gotten.
- All runnable tasks are sorted in the rbtree by the "rg->fair_clock - p->wait_runtime" key.
  - wait_runtime > 0 when inserted
  - CFS picks the 'leftmost' task and sticks to it

More CFS

- In practice it works like this:
  - the system runs a task a bit, and when the task schedules (or a scheduler tick happens) the task's CPU usage is 'accounted for':
    - When this task gets scheduled to the CPU, its wait_runtime value starts decrementing.
    - As this value falls to such a level that other tasks become the new left-most task of the red-black tree and the current one gets preempted
    - Newly woken tasks are put into the tree more and more to the right
  - slowly but surely giving a chance for every task to become the 'leftmost task' and thus get on the CPU within a deterministic amount of time.

Linux Process Structure

- called a "task_struct"
- (see sched.h ~ line 917)
- Key points:
  - state field
  - lists: identified with "struct list_head"

Data types in C

- structures: like a class without functions

```
struct node {
    int value;
    char name[10];
};
struct node node_instance;
struct node * ptr_to_node;
```

- typedef: give a name to another type

```
typedef struct node node_type;
typedef struct node * ptr_node_type;
```

- Recursive structures

```
struct node {
    int value;
    struct node * next;
};
```


Data Structures in C – Linked List

- Linked Lists

```
typedef struct node_s {
  what ever;
  struct node_s *next;
} node, *node_ptr;

node_ptr my_ptr, ptr;
struct header {
  node_ptr first, last;
} my_list;
```

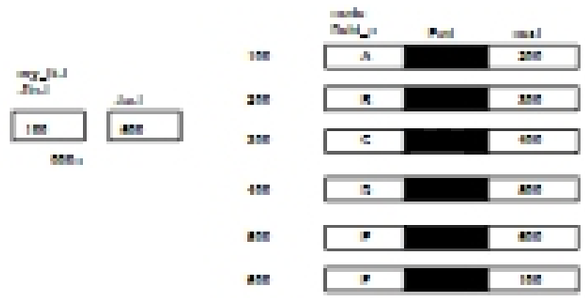

Allocating a List

```
int i;
/* Allocate the first list node */
my_ptr = (node_ptr)malloc( sizeof(node));
if (!my_ptr) exit(1);
my_list.first = my_ptr;
my_list.last = my_ptr;
my_ptr->letter = 'a';
my_ptr->next = my_list.first;  /* Make the list circular */
```

Allocating a List

```
/* allocate records 2 through 0 to the list */
for (i=0; i <= 0; i++)
{
  my_ptr = (node_ptr)malloc(sizeof(node));
      if(!my_ptr) exit(1);
  ptr = my_list.last;          // find end of list
  my_list.last = my_ptr;
  ptr->next = my_ptr;          // update ptr in former last
  my_ptr->field_1 = ptr->field_1 + 1; // move to next char
  my_ptr->next = my_list.first; // make circular again
}
```


Layout in Memory



Freeing a list

```
while (my_list.first != NULL ) {
  my_ptr = my_list.first;
  my_list.first = my_ptr->next;
  if (my_list.last == my_ptr) {
    my_list.last = NULL;
    my_list.first = NULL;
  }
  free(my_ptr);
}
```


Linux Linked Lists

Use structure to embed:

```
struct list_head {
    struct list_head *next, *prev;
};
```

Initialize: make them point to themselves

```
void INIT_LIST_HEAD(struct list_head *list)
{
    list->next = list;
    list->prev = list;
}
```

Usage: embed in other structures

```
struct task_struct {
    struct list_head tasks;
    ...
}
```