

CS 537

Section 4: Data Structures in C

Michael Swift

© 2004-2007 Ed Lazowska, Hank Levy, Andrea and
Remzi Arpaci-Dusseau, Michael Swift

1

Linux Completely Fair Scheduler

- Written in 62 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 2 tasks running, after first runs, second is behind by equal amount of time.

© 2004-2007 Ed Lazowska, Hank Levy, Andrea and
Remzi Arpaci-Dusseau, Michael Swift

2

More CFS

- Fairness
 - Idea: because you waited longer, you deserve more CPU later
 - runqueue 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 runqueue, 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 runqueue.
 - rq->fair_clock values measure the 'expected CPU time' a task should have gotten.
- All runnable tasks are sorted in the rbtree by the "rq->fair_clock - p->wait_runtime" key,
 - wait_runtime = 0 when inserted
 - CFS picks the 'leftmost' task and sticks to it

© 2004-2007 Ed Lazowska, Hank Levy, Andrea and
Remzi Arpaci-Dusseau, Michael Swift

3

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.

© 2004-2007 Ed Lazowska, Hank Levy, Andrea and
Remzi Arpaci-Dusseau, Michael Swift

4

Linux Process Structure

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

© 2004-2007 Ed Lazowska, Hank Levy, Andrea and
Remzi Arpaci-Dusseau, Michael Swift

5

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;
};
```

© 2004-2007 Ed Lazowska, Hank Levy, Andrea and
Remzi Arpaci-Dusseau, Michael Swift

6

Data Structures in C – Linked List

- Linked Lists

```
typedef struct node_s {
    char field_n;
    struct node_s *next;
} node, *node_ptr;

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

© 2004-2007 Ed Lazowska, Hank Levy, Andrea and
Remzi Arpaci-Dusseau, Michael Swift

7

Allocating a List

```
int i;
/* allocate the first list record */
my_ptr = (node_ptr)malloc( sizeof(node));
if (!my_ptr) exit(1);
my_list.first = my_ptr;
my_list.last = my_ptr;
my_ptr->field_n = 'A';
my_ptr->next = my_list.first; /* makes the list circular */
```

© 2004-2007 Ed Lazowska, Hank Levy, Andrea and
Remzi Arpaci-Dusseau, Michael Swift

8

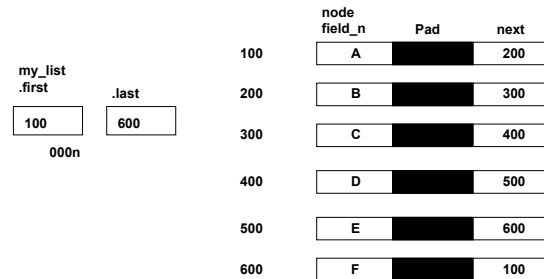
Allocating a List

```
/* allocate records 2 through 6 to the list */
for (i=2; i <= 6; 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_n = ptr->field_n + 1; // move to next char
    my_ptr->next = my_list.first; // make circular again
}
```

© 2004-2007 Ed Lazowska, Hank Levy, Andrea and
Remzi Arpaci-Dusseau, Michael Swift

9

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);
}
```

© 2004-2007 Ed Lazowska, Hank Levy, Andrea and
Remzi Arpaci-Dusseau, Michael Swift

11

Linux Linked Lists

List 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;
    ...
}
```

© 2004-2007 Ed Lazowska, Hank Levy, Andrea and
Remzi Arpaci-Dusseau, Michael Swift

12

List Functions

Add to a list:

```
void __list_add(struct list_head *new, struct list_head *prev, struct list_head *next) {
    next->prev = new;
    new->next = next;
    new->prev = prev;
    prev->next = new;
}
void list_add(struct list_head *new, struct list_head *head) {
    __list_add(new, head, head->next);
}
```

Remove from a list

```
void __list_del(struct list_head * prev, struct list_head * next) {
    next->prev = prev;
    prev->next = next;
}
void list_del(struct list_head *entry) {
    __list_del(entry->prev, entry->next);
}
```

© 2004-2007 Ed Lazowska, Hank Levy, Andrea and
Remzi Arpaci-Dusseau, Michael Swift

13

Using a linked list

- struct task_struct:
 - run_list, children, siblings
 - struct list_head tasks;
- kernel/fork.c: copy_process()
 - list_add(&p->tasks, &init_task.tasks);
 - Adds to global list of tasks
- kernel/exit: unhash_process()
 - list_del(&p->tasks);
 - deletes from global list

© 2004-2007 Ed Lazowska, Hank Levy, Andrea and
Remzi Arpaci-Dusseau, Michael Swift

14

List iteration

- For loop

```
#define list_for_each(pos, head) \
    for (pos = (head)->next; prefetch(pos->next), pos != (head); \
         pos = pos->next)
```

You need to use list_entry here

```
#define list_for_each_entry(pos, head, member) \
    for (pos = list_entry((head)->next, typeof(*pos), member); \
         prefetch(pos->member.next), &pos->member != (head); \
         pos = list_entry(pos->member.next, typeof(*pos), member))
```

This version provides the list_entry() internally

- Example:

```
list_for_each_entry(mod, &modules, list) {
    if (strcmp(mod->name, name) == 0)
        return mod;
}
```

© 2004-2007 Ed Lazowska, Hank Levy, Andrea and
Remzi Arpaci-Dusseau, Michael Swift

15

Memory Manipulation Functions

Header File:

<string.h>

```
void *memcpy (void *dest, const void *src, size_t n);
void *memmove(void *dest, const void *src, size_t n);
int memcmp (const void *s1, const void *s2, size_t n);
void *memset (void *s, int c, size_t n);
```

Memory Manipulation Functions

```
/* memcpy.c: memcpy example */
#include <stdio.h>
#include <string.h>

void main(void)
{
    char src[ ] = "*****";
    char dest[ ] = "abcdefghijklmnopqrstuvwxy01234";
    char *ptr;
    printf("destination before memcpy: %s\n", dest);
    ptr = (char *) memcpy(dest, src, strlen(src));
    if (ptr)
        printf("destination after memcpy: %s\n", dest);
    else
        printf("memcpy failed\n"); }
}
```

Memory Manipulation Functions

```
/* memmove.c memmove example */

#include <string.h>
#include <stdio.h>

void main(void)
{
    char dest [80] = "abcdefghijklmnopqrstuvwxy012345" ;

    printf ( "dest prior to memmove: %s\n", dest);
    memmove (&dest[5], &dest[26], 6);
    printf ("dest after memmove: %s\n", dest);
}
}
```

Memory Manipulation Functions

```
/* memset.c: memset example */

#include <string.h>
#include <stdio.h>
#include <mem.h>

void main(void)
{
    char buffer[ ] = "Hello world\n";

    printf("Buffer before memset: %s\n", buffer);
    memset(buffer, '*', strlen(buffer) - 1);
    printf("Buffer after memset: %s\n", buffer);
}
}
```

Memory Manipulation Functions

```
char *buf1 = "aaa", *buf2 = "bbb", *buf3 = "ccc";
int stat;
stat = memcmp(buf2, buf1, strlen(buf2));
if (stat > 0)
    printf("buffer 2 is greater than buffer 1\n");
else
    printf("buffer 2 is less than buffer 1\n");
stat = memcmp(buf2, buf3, strlen(buf2));
if (stat > 0)
    printf("buffer 2 is greater than buffer 3\n");
else
    printf("buffer 2 is less than buffer 3\n");
}
```