Paul Prince's MPX

R1

Generated by Doxygen 1.7.3

Thu Mar 17 2011 21:00:47

Contents

Chapter 1

Introduction

1.1 Repository

Version-control information is managed by Git, and hosted by GitHub: https://github.com/pprince/cs450

1.2 Documentation

Documentation for developers is generated by Doxygen; for detailed information about the files, functions, data structures, etc. that make up MPX and how they relate to each other, refer to:

- "MPX Programmer's Manual"

which can be found in the doc/ directory. Also, in the same directory, you can find the current version of:

- "MPX User's Manual"

Todo

Generally, documentation is incomplete.

Todo

Generally, we need to make lines break cleanly at 80-columns; Doxygen forces such line-breaks on us in the LaTeX output, but our source code frequently uses longer lines (making the PDF version of the developer manual very ugly!

Chapter 2

Todo List

Global find_pcb(char ∗name) This really should be done a little cleaner, possibly using a foreach() macro, like the one at: `http://stackoverflow.com/questions/400951/c-foreach-or`

page Introduction Generally, documentation is incomplete.

Generally, we need to make lines break cleanly at 80-columns; Doxygen forces such line-breaks on us in the LaTeX output, but our source code frequently uses longer lines (making the PDF version of the developer manual very ugly!

File mpx_cmds.c We should typedef structs (particularly struct mpx_command).

Chapter 3

Bug List

Global add_command(char ∗name, void(∗function)(int argc, char ∗argv[])) This function doesn't check for failure to allocate memory for the new command struct.

Global mpx_shell(void) A command should be able to depend on argv[argc] == NULL, but we do not currently implement this feature.

Chapter 4

Data Structure Documentation

4.1 date_rec Struct Reference

Data Fields

- int **month**
- int **day**
- int **year**

The documentation for this struct was generated from the following file:

- mpx/mpx_supt.h

4.2 mpx_command Struct Reference

Node type for a singly-linked list of MPX commands.

```
#include <mpx_cmds.h>
```

Data Fields

- char ∗ **name**
- void(∗ **function**)(int argc, char ∗argv[ ])
- struct mpx_command ∗ **next**

4.2.1 Detailed Description

Node type for a singly-linked list of MPX commands.

The documentation for this struct was generated from the following file:

- mpx/mpx_cmds.h

4.3 params Struct Reference

Data Fields

- int **op_code**
- int **device_id**
- char ∗ **buf_p**
- int ∗ **count_p**

The documentation for this struct was generated from the following file:

- mpx/mpx_supt.c

4.4 pcb_queue_node_t Struct Reference

Data Fields

- struct pcb_queue_node ∗ next

  Pointer to the next PCB node in the queue.

- struct pcb_queue_node ∗ prev

  Pointer to the previous PCB node in the queue.

- pcb_t ∗ pcb

  Pointer to the actual PCB associated with this node.

4.4.1 Field Documentation

4.4.1.1 struct pcb_queue_node∗ **next**

Pointer to the next PCB node in the queue.

4.4.1.2 struct pcb_queue_node∗ prev

Pointer to the previous PCB node in the queue.

4.4.1.3 pcb_t∗ pcb

Pointer to the actual PCB associated with this node.

The documentation for this struct was generated from the following file:

- mpx/pcb.h

4.5 pcb_queue_t Struct Reference

PCB queue; represents a queue of processes.

```
#include <pcb.h>
```

Data Fields

- pcb_queue_node_t ∗ head

  Pointer to the first element in the queue.

- pcb_queue_node_t ∗ tail

  Pointer to the last element in the queue.

- unsigned int length

  Number of elements in the queue.

- pcb_queue_sort_order_t sort_order

  Specifies how elements in this queue are sorted at insert-time.

4.5.1 Detailed Description

PCB queue; represents a queue of processes.

4.5.2 Field Documentation

4.5.2.1 pcb_queue_node_t∗ head

Pointer to the first element in the queue.

4.5.2.2 pcb_queue_node_t* tail

Pointer to the last element in the queue.

4.5.2.3 unsigned int length

Number of elements in the queue.

4.5.2.4 pcb_queue_sort_order_t sort_order

Specifies how elements in this queue are sorted at insert-time.

The documentation for this struct was generated from the following file:

- mpx/pcb.h

4.6 pcb_t Struct Reference

Process control block structure.

```
#include <pcb.h>
```

Data Fields

- char name [MAX_ARG_LEN+1]

  Name of the process (i.e., its argv[0] in unix-speak).

- process_class_t class

  Process class (differentiates applications from system processes.

- int priority

  Process priority.

- process_state_t state

  Process state (Ready, Running, or Blocked).

- unsigned char * stack_top

  Pointer to the top of this processes's stack.

- unsigned char * stack_base

  Pointer to the bottom of this processes's stack.

- int memory_size

  Memory size ...

- unsigned char ∗ load_address

  Load address ...

- unsigned char ∗ exec_address

  Execution address ...

4.6.1 Detailed Description

Process control block structure.

4.6.2 Field Documentation

4.6.2.1 char **name**[MAX_ARG_LEN+1]

Name of the process (i.e., its argv[0] in unix-speak).

4.6.2.2 **process_class_t class**

Process class (differentiates applications from system processes.

4.6.2.3 int **priority**

Process priority.

Higher numerical value = higher priority.

Valid values are -128 through 127 (inclusive).

4.6.2.4 **process_state_t state**

Process state (Ready, Running, or Blocked).

4.6.2.5 unsigned char∗ **stack_top**

Pointer to the top of this processes's stack.

4.6.2.6 unsigned char∗ stack_base

Pointer to the bottom of this processes's stack.

4.6.2.7 int memory_size

Memory size ...

will be used in R3 and R4.

4.6.2.8 unsigned char∗ load_address

Load address ...

will be used in R3 and R4.

4.6.2.9 unsigned char∗ exec_address

Execution address ...

will be used in R3 and R4.

The documentation for this struct was generated from the following file:

- mpx/pcb.h

Chapter 5

File Documentation

5.1 mpx/mpx.c File Reference

MPX main() function.

```
#include "mpx_supt.h"
#include "mpx_util.h"
#include "mpx_sh.h"
#include "mpx_cmds.h"
#include "pcb.h"
```

Functions

- void main (int argc, char ∗argv[ ])
  This is the start-of-execution for the MPX executable.

5.1.1 Detailed Description

MPX main() function.

Author

Paul Prince <paul@littlebluetech.com>

Date

2011

This file contains the start-of-execution, i.e. function main(), for MPX, and also the top-level Doxygen documentation that becomes the introductory sections of the developer's manual.

5.1.2 Function Documentation

5.1.2.1 void main (int *argc,* char ∗ *argv[]*)

This is the start-of-execution for the MPX executable.

```
{
        sys_init( MODULE_R1 );  /* System-specific initialization.       */

        init_commands();        /* Initialization for MPX user commands. */
        init_pcb_queues();      /* Initialization for PCB queues.        */

        mpx_shell();            /* Execute the command-handler loop.     */

        /* mpx_shell() should never return, so if we get here, then
         * we should exit with error status (but don't actually...). */
        printf("FATAL ERROR: mpx_shell() returned! That shouldn't happen...\n");
        sys_exit();     /* Terminate, after doing MPX-specific cleanup. */
}
```

5.2 mpx/mpx_cmds.c File Reference

MPX shell commands (help, ls, exit, etc.)

```
#include "mpx_cmds.h"
#include "mpx_supt.h"
#include "mpx_util.h"
#include "pcb.h"
#include <string.h>
```

Functions

- void add_command (char ∗name, void(∗function)(int argc, char ∗argv[ ]))

  Adds a command to the MPX shell.

- void dispatch_command (char ∗name, int argc, char ∗argv[ ])

  Runs the shell command specified by the user, if it is valid.

- void **mpxcmd_commands** (int argc, char ∗argv[ ])

- void mpxcmd_date (int argc, char ∗argv[ ])
- void **mpxcmd_exit** (int argc, char ∗argv[ ])
- void **mpxcmd_help** (int argc, char ∗argv[ ])
- void **mpxcmd_version** (int argc, char ∗argv[ ])
- void **mpxcmd_ls** (int argc, char ∗argv[ ])
- void mpxcmd_suspend (int argc, char ∗argv[ ])

  Implements the `suspend` *shell command.*

- void mpxcmd_resume (int argc, char ∗argv[ ])

  Implements the `resume` *shell command.*

- void mpxcmd_renice (int argc, char ∗argv[ ])

  Implements the `renice` *shell command.*

- void mpxcmd_ps (int argc, char ∗argv[ ])

  Implements the `ps` *shell command.*

- void mpxcmd_create_pcb (int argc, char ∗argv[ ])

  Implements the `create_pcb` *shell command.*

- void mpxcmd_delete_pcb (int argc, char ∗argv[ ])

  Implements the `delete_pcb` *shell command.*

- void mpxcmd_block (int argc, char ∗argv[ ])

  Implements the `block` *shell command.*

- void mpxcmd_unblock (int argc, char ∗argv[ ])

  Implements the `unblock` *shell command.*

- void **init_commands** (void)

Variables

- static struct mpx_command ∗ list_head = NULL

  A linked-list of MPX shell commands.

5.2.1 Detailed Description

MPX shell commands (help, ls, exit, etc.)

Author

Paul Prince <paul@littlebluetech.com>

Date

2011

This file implements each of the user commands for MPX.

Todo

We should typedef structs (particularly struct mpx_command).

5.2.2 Function Documentation

5.2.2.1 void add_command (char * *name*, void(*)(int argc, char *argv[]) *function*)

Adds a command to the MPX shell.

Bug

This function doesn't check for failure to allocate memory for the new command struct.

Parameters

in	*name*	The command name that will be made available in the shell.
in	*function*	The C function which will implement the shell command.

```
{
        /* Temporary variable for iterating through the list of commands. */
        struct mpx_command *this_command;

        /* Allocate space for the new command structure. */
        struct mpx_command *new_command =
                (struct mpx_command *)sys_alloc_mem(sizeof(struct mpx_command));
        new_command->name = (char *)sys_alloc_mem(MAX_ARG_LEN+1);
        /* Initialize the structure. */
        strcpy( new_command->name, name );
        new_command->function = function;
        new_command->next = NULL;

        /* Insert the new command into the linked-list of commands. */
        this_command = list_head;
        if ( this_command == NULL ) {
                list_head = new_command;
        } else {
                while ( this_command->next != NULL ){
                        this_command = this_command->next;
                }
                this_command->next = new_command;
        }
}
```

5.2.2.2 void dispatch_command (char ∗ *name,* int *argc,* char ∗ *argv[]*)

Runs the shell command specified by the user, if it is valid.

This function checks to see if the shell command given unabiguously matches a valid MPX shell command, and if so, runs that command (passing the provided argc and argv through).

This dispatcher allows abbreviated commands; if the requested command matches multiple (or zero) valid MPX shell commands, the user is alerted.

Attention

Produces output (via printf)!

```
                                                      {
  /* Temporary variable for iterating through the list of commands. */
  struct mpx_command *this_command = list_head;

  /* Temporary variables to keep track of matching command names. */
  int num_matches = 0;
  struct mpx_command *first_match;

  /* Iterate through the linked list of commands, */
  while( this_command != NULL ) {

          /* Check to see if the given command is a valid abbrev. for the c
urrent command from the list */
          if( strncmp( this_command->name, name, strlen(name) ) == 0 ) {
                  /* If so, keep track of how many matches thus far, */
                  num_matches++;
                  if (num_matches == 1) {
                          /* This is the first match in the list for the gi
ven command. */
                          first_match = this_command;
                  } else if (num_matches == 2) {
                          /* This is the first duplicate match in the list;

                           * Print out the 'ambiguous command' header,
                           * plus the first AND current ambiguous commands.
 */
                          printf("Ambiguous command: %s\n", name);
                          printf("    Matches:\n");
                          printf("        %s\n", first_match->name);
                          printf("        %s\n", this_command->name);
                  } else {
                          /* This is a subsequent duplicate match;
                           * by this time, the header etc. has already been
```

```
       printed,
                                 * so we only need to print out the current comma
      nd name. */
                                printf("        %s\n", this_command->name);
                        }
                }

                this_command = this_command->next;
        }

        /* If we got a command name that matches unambiguously, run that command.
       */
        if ( num_matches == 1 ){
                first_match->function(argc, argv);
        }

        /* Otherwise, if we got no matches at all, say so. */
        if ( num_matches == 0 ){
                printf("ERROR: Invalid command name.\n");
                printf("Type \"commands\" to see a list of valid commands.\n");
        }
}
```

5.2.2.3 void mpxcmd_date (int *argc*, char ∗ *argv[]*)

< Temp. storage for the return value of sys_ functions.

< Structure to hold a date (day, month, and year). Will be used for both getting and setting the MPX system date.

```
                                           {
        int retval;
        date_rec date;

        if ( argc == 1 ){
                sys_get_date(&date);
                printf("Current MPX system date (yyyy-mm-dd): %04d-%02d-%02d\n",
      date.year, date.month, date.day);
                return;
        }

        if ( argc == 4 ){

                date.year  = atoi(argv[1]);
                date.month = atoi(argv[2]);
                date.day   = atoi(argv[3]);

                if ( ! mpx_validate_date(date.year, date.month, date.day) ) {
                        printf("ERROR: Invalid date specified; MPX system date is
       unchanged.\n");
                        printf("       Valid dates are between 1900-01-01 and 299
      9-12-31, inclusive.\n");
                        return;
                }
                retval = sys_set_date(&date);
                if ( retval != 0 ) {
                        printf("ERROR: sys_set_date() returned an error.\n");
                        return;
                }

                printf("The MPX system date has been changed.\n");
                return;
        }

        printf("ERROR: Wrong number of arguments to ’date’.\n");
        printf("       Type ’help date’ for usage information.\n");
}
```

5.2.2.4 void mpxcmd_create_pcb (int *argc*, char * *argv[]*)

Implements the `create_pcb` shell command.

Attention

This TEMPORARY command will be replaced later.

```
{
        pcb_t           *new_pcb;
        int             new_pcb_priority;
        process_class_t new_pcb_class;
        pcb_queue_t     *new_pcb_dest_queue;

        if ( argc != 4 ){
                printf("ERROR: Wrong number of arguments to create_pcb.\n");
                return;
        }

        if ( strlen(argv[1]) > MAX_ARG_LEN ) {
                printf("ERROR: Specified process name is too long.\n");
                return;
        }

        new_pcb_priority = atoi(argv[3]);

        if ( new_pcb_priority < -127 || new_pcb_priority > 128 ){
                printf("ERROR: Invalid priority specified.\n");
                printf("Priority must be between -127 and 128 (inclusive).\n");
                return;
        }

        if ( strlen(argv[2]) == 1 && argv[2][0] == ’A’ ) {
                new_pcb_class = APPLICATION;
        } else if ( strlen(argv[2]) == 1 && argv[2][0] == ’S’ ) {
                new_pcb_class = SYSTEM;
        } else {
                printf("ERROR: Invalid process class specified.\n");
                return;
        }

        new_pcb = setup_pcb( argv[1], new_pcb_priority, new_pcb_class);

        if ( new_pcb == NULL ){
                printf("ERROR: Failure creating process.\n");
                return;
        }

        new_pcb_dest_queue = insert_pcb( new_pcb );

        if ( new_pcb_dest_queue == NULL ){
                printf("ERROR: Failure enqueuing new process.\n");
        }

        printf("Success: Process created.\n");
}
```

5.2.2.5 void mpxcmd_delete_pcb (int *argc*, char ∗ *argv[]*)

Implements the `delete_pcb` shell command.

Attention

This TEMPORARY command will be replaced later.

```
{
}
```

5.2.2.6 void mpxcmd_block (int *argc*, char ∗ *argv[]*)

Implements the `block` shell command.

Attention

This TEMPORARY command will be replaced later.

```
{
}
```

5.2.2.7 void mpxcmd_unblock (int *argc*, char ∗ *argv[]*)

Implements the `unblock` shell command.

Attention

This TEMPORARY command will be replaced later.

```
{
}
```

5.2.3 Variable Documentation

5.2.3.1 struct mpx_command∗ list_head = NULL [static]

A linked-list of MPX shell commands.

5.3 mpx/mpx_sh.c File Reference

MPX Shell, aka Command Handler.

```
#include "mpx_sh.h"
#include "mpx_supt.h"
#include "mpx_util.h"
#include "mpx_cmds.h"
#include <string.h>
```

Functions

- void mpx_setprompt (char ∗new_prompt)

  Sets the current prompt to whatever string is given.

- void mpx_shell (void)

  This function implements the MPX shell (command-line user interface).

Variables

- static char ∗ mpx_prompt_string = NULL

  The current prompt string.

5.3.1 Detailed Description

MPX Shell, aka Command Handler. This file implements the user interface for MPX.

5.3.2 Function Documentation

5.3.2.1 void mpx_setprompt (char * *new_prompt*)

Sets the current prompt to whatever string is given.

If new_prompt is NULL, this is a no-op.

```
                                    {
        if (new_prompt == NULL) return;
        if (mpx_prompt_string != NULL) {
                sys_free_mem(mpx_prompt_string);
        }
        mpx_prompt_string = (char *)sys_alloc_mem(strlen(new_prompt)+1);
        strcpy(mpx_prompt_string, new_prompt);
}
```

5.3.2.2 void mpx_shell (void)

This function implements the MPX shell (command-line user interface).

mpx_shell() never returns!

Bug

A command should be able to depend on argv[argc] == NULL, but we do not currently implement this feature.

```
                     {

        /* A buffer to hold the command line input by the user.
         * We include space for the \r, \n, and \0 characters, if any. */
        char cmdline[ MAX_CMDLINE_LEN+2 ];

        /* Buffer size argument for passing to sys_req(). */
        int line_buf_size = MAX_CMDLINE_LEN;

        /* Used to capture the return value of sys_req(). */
        int err;

        /* argc to be passed to MPX command; works just like the one passed to ma
      in(). */
        int argc;
        /* argv array to be passed to MPX command; works almost just like the one
       passed to main().
         *
         * But there is one caveat: argv[argc] is undefined in my implementation,
       not garanteed to be NULL. */
        char **argv;

        /* Temporary pointer for use in string tokenization. */
        char *token;
```

```
  /* Delimiters that separate arguments in the MPX shell command-line envir
onment. */
  char *delims = "\t \n";

  /* An index for use in for(;;) loops. */
  int i;
  /* An index for use in nested for(;;) loops. */
  int j;

  /* We must initialize the prompt string. */
  mpx_setprompt(MPX_DEFAULT_PROMPT);

  /* Loop Forever; this is the REPL. */
  /* This loop terminates only via the MPX 'exit' command. */
  for(;;) {
          /* Output the current MPX prompt string. */
          printf("%s", mpx_prompt_string);

          /* Read in a line of input from the user. */
          sys_req( READ, TERMINAL, cmdline, &line_buf_size );

          /* Remove trailing newline. */
          mpx_chomp(cmdline);

          /* Allocate space for the argv argument that is to be sent to an
MPX command. */
          argv = (char **)sys_alloc_mem( sizeof(char**) * (MAX_ARGS+1) ); /
* +1 for argv[0] */
          for( i=0; i < MAX_ARGS+1; i++ ){                                /
* +1 for argv[0] */
                  argv[i] = sys_alloc_mem(MAX_ARG_LEN+1);                 /
* +1 for \0 */
          }


          /* Tokenize the command line entered by the user, and set argc. *
/
          /* 0 is a special value here for argc; a value > 0 after the for
loop indicates
           * that tokenizing was successful and that argc and argv contain
valid data.
           *
           *****  NOTE:  argc includes argv[0], but MAX_ARGS does not!  ***
**/

          argc = 0; token = NULL;

          for( i=0; i < MAX_ARGS+1; i++ ){

                  if (i==0) {
                          token = strtok( cmdline, delims );
                  } else {
                          token = strtok( NULL, delims );
                  }

                  if (token == NULL) {
```

```
                                /* No more arguments. */
                                break;
                        }

                        if (strlen(token) > MAX_ARG_LEN) {
                                /* This argument is too long. */
                                printf("ERROR: Argument too long. MAX_ARG_LEN is
      %d.\n", MAX_ARG_LEN);
                                argc = 0;
                                break;
                        }

                        argc++;
                        strcpy( argv[i], token );
                }

                if ( strtok( NULL, delims ) != NULL ){
                        /* Too many arguments. */
                        printf("ERROR: Too many arguments. MAX_ARGS is %d.\n", MA
      X_ARGS);
                        continue;
                }

                if ( argc <= 0 ) {
                        /* Blank command; just re-print the prompt. */
                        continue;
                }

                /* Run the command, or print an error if it is invalid. */
                dispatch_command( argv[0], argc, argv );

                /* Free the memory for the dynamically-allocated *argv[] */
                for( i=0; i < MAX_ARGS+1; i++ ){
                        sys_free_mem( argv[i] );
                }
                sys_free_mem( argv );
        }
}
```

5.3.3 Variable Documentation

5.3.3.1 char∗ **mpx_prompt_string** = NULL [static]

The current prompt string.

5.4 mpx/mpx_util.c File Reference

Various utility functions used by all of MPX.

```
#include "mpx_util.h"
```

```
#include "mpx_supt.h"
```

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

Functions

- int mpx_chomp (char ∗str)

  Removes trailing newline, if any.

- int **mpx_validate_date** (int year, int month, int day)
- int **mpx_cat** (char ∗file_name)

5.4.1 Detailed Description

Various utility functions used by all of MPX. This file contains the functions etc. to implement the user interface for MPX.

5.4.2 Function Documentation

5.4.2.1 int mpx_chomp (char ∗ *str*)

Removes trailing newline, if any.

This function checks to see if the last character in a string is a newline, and, if so, removes it. Otherwise, the string is left unchanged.

The input must be a valid (allocated and null-terminated) C string, otherwise the results are undefined (but will most likley result in a segmentation fault / protection fault).

Returns the number of characters removed from the string.

Parameters

str	The string to chomp.

```
                            {
        if( strlen(str) > 0 ){
                if( str[ strlen(str)-1 ] == '\n' ){
                        str[ strlen(str)-1 ] = '\0';
                        return 1;
                }
        }
        return 0;
}
```

5.5 mpx/pcb.c File Reference

PCBs, process queues, and functions to operate on them.

```
#include "pcb.h"
#include "mpx_supt.h"
#include "mpx_util.h"
```

Functions

- void init_pcb_queues (void)

  Must be called before using any other PCB or queue functions.

- pcb_t ∗ allocate_pcb (void)

  Allocates memory for a new PCB, but does not initialize it.

- void free_pcb (pcb_t ∗pcb)

  De-allocates the memory that was used for a PCB.

- pcb_t ∗ setup_pcb (char ∗name, int priority, process_class_t class)

  Creates, allocates, and initializes a new PCB object.

- pcb_t ∗ find_pcb_in_queue (char ∗name, pcb_queue_t ∗queue)

  Search the given queue for the named process.

- pcb_t ∗ find_pcb (char ∗name)

  Finds a process.

- pcb_queue_t ∗ remove_pcb (pcb_t ∗pcb)

  Removes a PCB from its queue.

- pcb_queue_t ∗ insert_pcb (pcb_t ∗pcb)

  Inserts a PCB into the appropriate queue.

Variables

- static pcb_queue_t **queue_ready**
- static pcb_queue_t **queue_blocked**
- static pcb_queue_t **queue_susp_ready**
- static pcb_queue_t **queue_susp_blocked**
- pcb_queue_t ∗ **queues** [4]

5.5.1 Detailed Description

PCBs, process queues, and functions to operate on them.

Author

Paul Prince <paul@littlebluetech.com>

Date

2011

5.5.2 Function Documentation

5.5.2.1 void init_pcb_queues (void)

Must be called before using any other PCB or queue functions.

```
{
        queues[0] = &queue_ready;
        queue_ready.head                = NULL;
        queue_ready.tail                = NULL;
        queue_ready.length              = 0;
        queue_ready.sort_order          = PRIORITY;

        queues[1] = &queue_blocked;
        queue_blocked.head              = NULL;
        queue_blocked.tail              = NULL;
        queue_blocked.length            = 0;
        queue_blocked.sort_order        = FIFO;

        queues[2] = &queue_susp_ready;
        queue_susp_ready.head           = NULL;
        queue_susp_ready.tail           = NULL;
        queue_susp_ready.length         = 0;
        queue_susp_ready.sort_order     = FIFO;

        queues[3] = &queue_susp_blocked;
        queue_susp_blocked.head         = NULL;
        queue_susp_blocked.tail         = NULL;
        queue_susp_blocked.length       = 0;
        queue_susp_blocked.sort_order   = FIFO;
}
```

5.5.2.2 pcb_t∗ allocate_pcb (void)

Allocates memory for a new PCB, but does not initialize it.

This function will also allocate memory for the PCB's stack, and initialize the stack_top and stack_base members.

Returns

Returns a pointer to the new PCB, or NULL if an error occured.

```
{
        /* Pointer to the new PCB we will allocate. */
        pcb_t *new_pcb;

        /* Allocate memory for the PCB. */
        new_pcb = (pcb_t *)sys_alloc_mem(sizeof(pcb_t));
        if ( new_pcb == NULL ) {
                /* Error allocating memory for the PCB. */
                return NULL;
        }

        /* Allocate memory for the PCB’s stack. */
        new_pcb->stack_base = (unsigned char *)sys_alloc_mem(STACK_SIZE);
        if ( new_pcb->stack_base == NULL ) {
                /* Error allocating memory for the PCB’s stack. */
                sys_free_mem(new_pcb);
                return NULL;
        }

        /* Initialize stack_top member. */
        new_pcb->stack_top = new_pcb->stack_base + STACK_SIZE;


        return new_pcb;
}
```

5.5.2.3 void free_pcb (pcb_t * *pcb*)

De-allocates the memory that was used for a PCB.

```
{
        sys_free_mem(pcb->stack_base);
        sys_free_mem(pcb);
}
```

5.5.2.4 pcb_t* setup_pcb (char * *name*, int *priority*, process_class_t *class*)

Creates, allocates, and initializes a new PCB object.

This function creates a new PCB object (pcb_t), then calls allocate_pcb() to do the allocation step. It then initializes the PCB’s various fields according to both default values and the parameters passed in.

Returns

Returns a pointer to the new PCB, or NULL if an error occured.

Parameters

name	Name of the new process. Must be unique among all processes.
priority	Priority of the process. Must be between -127 and 128 (incl.)
class	Class of the process; one of APPLICATION or SYSTEM.

```
{
        /* Loop index. */
        int i;

        /* Pointer to the new PCB we're creating. */
        pcb_t *new_pcb;

        /* Check that arguments are valid. */
        if ( find_pcb(name) != NULL ) {
                /* Name is not unique. */
                return NULL;
        }
        if ( strlen(name) > MAX_ARG_LEN || name == NULL ) {
                /* Invalid name. */
                return NULL;
        }
        if ( priority < -127 || priority > 128 ) {
                /* Value of priority is out of range. */
                return NULL;
        }
        if ( class != APPLICATION && class != SYSTEM ) {
                /* Invalid class specified. */
                return NULL;
        }


        /* Allocate the new PCB. */
        new_pcb = allocate_pcb();
        if (new_pcb == NULL) {
                /* Allocation error. */
                return NULL;
        }


        /* Set the given values. */
        new_pcb->priority       = priority;
        new_pcb->class          = class;
        strcpy( new_pcb->name, name );


        /* Set other default values. */
        new_pcb->state          = READY;
        new_pcb->memory_size    = 0;
        new_pcb->load_address   = NULL;
        new_pcb->exec_address   = NULL;

        /* Initialize the stack to 0's. */
        for (i=0; i<STACK_SIZE; i++) {
                *(new_pcb->stack_base + i) = (unsigned char)0;
        }
        return new_pcb;
}
```

5.5.2.5 pcb_t∗ find_pcb_in_queue (char ∗ *name*, pcb_queue_t ∗ *queue*) [private]

Search the given queue for the named process.

Returns

Returns a pointer to the PCB, or NULL if not found or error.

Parameters

name	The name of the process to find.
queue	The PCB queue in which to search for the process.

```
{
        pcb_queue_node_t *this_queue_node = queue->head;

        while (this_queue_node != NULL) {
                if ( strcmp( this_queue_node->pcb->name, name) == 0 ) {
                        return this_queue_node->pcb;
                }
                this_queue_node = this_queue_node->next;
        }

        /* If we get here, we didn't find the process. */
        return NULL;
}
```

5.5.2.6 pcb_t∗ find_pcb (char ∗ *name*)

Finds a process.

Searches all process queues.

Returns

Returns a pointer to the PCB, or NULL if not found or error.

Todo

This really should be done a little cleaner, possibly using a foreach() macro, like the one at: http://stackoverflow.com/questions/400951/c-foreach-or-similar

Parameters

name	The name of the process to find.

```
{
        /* Pointer to the requested PCB, if we find it. */
        pcb_t *found_pcb;

        /* Validate arguments. */
        if ( name == NULL || strlen(name) > MAX_ARG_LEN ) {
                /* Invalid process name. */
                return NULL;
        }

        /* Search for the PCB.  If we find it, return it. */
        if ( found_pcb = find_pcb_in_queue( name, &queue_ready ) ) {
                return found_pcb;
        }
        if ( found_pcb = find_pcb_in_queue( name, &queue_blocked ) ) {
                return found_pcb;
        }
        if ( found_pcb = find_pcb_in_queue( name, &queue_susp_ready ) ) {
                return found_pcb;
        }
        if ( found_pcb = find_pcb_in_queue( name, &queue_susp_blocked ) ) {
                return found_pcb;
        }

        /* If we get here, the process was not found. */
        return NULL;
}
```

5.5.2.7 pcb_queue_t* remove_pcb (pcb_t * *pcb*)

Removes a PCB from its queue.

Given a pointer to a valid and en-queued PCP, this function will remove that PCB from the queue that it is in.

However, this function will *not modify* the state member of the PCB; the caller is responsible for doing that, if the PCB is to be re-enqueued rather than de-allocated.

Returns

Returns a pointer to the new PCB, or NULL if an error occured.

Parameters

pcb	Pointer to the PCB to be de-queued.

```
{

}
```

5.5.2.8 pcb_queue_t* insert_pcb (pcb_t * *pcb*)

Inserts a PCB into the appropriate queue.

Inspects the PCB's state member to determine which queue to insert into.

Inspects the queue's sort_order member to determine whether to insert in order of priority, or to simply insert the PCB at the end of of the queue.

Parameters

pcb	Pointer to the PCB to be enqueued.

```
{
        /* Pointer to the queue we will insert into. */
        pcb_queue_t             *queue;
        /* Pointer to the new queue node descriptor we must make. */
        pcb_queue_node_t        *new_queue_node;
        /* For use in loops that iterating through the queue. */
        pcb_queue_node_t        *iter_node;

        /* Validate argument */
        if (pcb == NULL) {
                /* PCB to insert cannot be null... come on :) */
                return NULL;
        }

        /* Determine which queue we will insert this PCB into. */
        switch (pcb->state) {
                case READY:
                        queue = &queue_ready;
                break;
                case BLOCKED:
                        queue = &queue_blocked;
                break;
                case SUSP_READY:
                        queue = &queue_susp_ready;
                break;
                case SUSP_BLOCKED:
                        queue = &queue_susp_blocked;
                break;
                default:
                        /* Unexpected value for PCB state (maybe Running?) */
                        return NULL;
                break;
        }

        /* Allocate the new queue descriptor. */
        new_queue_node =
                (pcb_queue_node_t *)sys_alloc_mem(sizeof(pcb_queue_node_t));
        if ( new_queue_node == NULL ){
                /* Error allocating memory. */
                return NULL;
        }
        /* Do the insert ... */
        /* ----------------- */

        new_queue_node->pcb = pcb;

        /* Case one: queue is empty. */
        if ( queue->length == 0 ){
                new_queue_node->next    = NULL;
                new_queue_node->prev    = NULL;
                queue->head             = new_queue_node;
                queue->tail             = new_queue_node;
                queue->length           = 1;
                return queue;
        }

        /* Case two: FIFO queue; we only need to insert at end. */
        if ( queue->sort_order == FIFO ){
                goto INSERT_AT_END;
        }

        /* The hard case: insert in priority-order. */
        iter_node = queue->head;
        while (iter_node != NULL) {
                if ( iter_node->pcb->priority < pcb->priority ){
                        /* Insert before iter_node */
                        new_queue_node->prev = iter_node->prev;
                        iter_node->prev->next = new_queue_node;
                        iter_node->prev = new_queue_node;
                        new_queue_node->next = iter_node;
                        if ( queue->head == iter_node ){
                                queue->head = new_queue_node;
                        }
                        queue->length++;
                        return queue;
                }
                iter_node = iter_node->next;
        }
        /* If we got this far, we need to do an insert-at-the-end. */


        INSERT_AT_END:
                new_queue_node->next    = NULL;
                new_queue_node->prev    = queue->tail;
                queue->tail->next       = new_queue_node;
                queue->tail             = new_queue_node;
                queue->length++;
                return queue;
}
```

5.6 mpx/pcb.h File Reference

PCBs, process queues, and functions to operate on them.

```
#include "mpx_util.h"
```

Data Structures

- struct pcb_t

  Process control block structure.

- struct pcb_queue_node_t
- struct pcb_queue_t

  PCB queue; represents a queue of processes.

Defines

- #define STACK_SIZE 1024

  Amount of stack space to allocate for each process (in bytes).

- #define foreach_listitem(item, list) for (item = list->head; item != NULL; item = item->next)

  Provides syntactic sugar for looping over the elements of a linked list.

Enumerations

- enum process_state_t {
  RUNNING, **READY**, **BLOCKED**, **SUSP_READY**,
  SUSP_BLOCKED }

  Type for variables that hold the state of a process.

- enum process_class_t { **APPLICATION**, **SYSTEM** }

  Type for variables that hold the class of a process.

- enum pcb_queue_sort_order_t { **FIFO**, **PRIORITY** }

  Enum constants for process sort order (i.e., queue insertion order.)

Functions

- void init_pcb_queues (void)

  Must be called before using any other PCB or queue functions.

- pcb_t ∗ setup_pcb (char ∗name, int priority, process_class_t class)

  Creates, allocates, and initializes a new PCB object.

- pcb_t ∗ find_pcb (char ∗name)

  Finds a process.

- pcb_queue_t ∗ insert_pcb (pcb_t ∗pcb)

  Inserts a PCB into the appropriate queue.

Variables

- pcb_queue_t ∗ **queues** []

5.6.1 Detailed Description

PCBs, process queues, and functions to operate on them.

Author

Paul Prince <paul@littlebluetech.com>

Date

2011

5.6.2 Define Documentation

5.6.2.1 #define STACK_SIZE 1024

Amount of stack space to allocate for each process (in bytes).

5.6.2.2 #define foreach_listitem(*item,* *list*) for (item = list->head; item != NULL; item = item->next)

Provides syntactic sugar for looping over the elements of a linked list.

This function makes it a little more readable when you want to loop over elements in a linked list, starting with the head. Will work on both singly- and doubly-linked lists.

If you wish to stop processing early, before iterating through the entire list, simply call break as if you were in a `for(;;){}` or `while()` loop.

In order to use this function on your list, the following requirements must be satisfied:

- You must declare the variable you pass as `item` yourself.

- The list parameter must be a pointer to a struct that has a member named head that is a pointer to the first item in the list.
- In the case that the list is empty (i.e., contains zero elements), then list->head must point to NULL.
- The item parameter *and* the list->head member must both be pointers to structs of the same type, and,
- That struct must have a member named next that is a pointer to the next item in the list.
- The next member of the last item in the list *must* point to NULL.

And also, while the following rules may not be strict requirements, it is *strongly* encouraged that you adhere to them:

- If, in a given execution of the loop body, you modify the list by adding, removing, moving, any list items, you should break out of the loop; *you should not,* having so-modified the list, continue on to the next iteration / execution of the loop body.
- You should not modify the value of item inside the loop body.

Note that you're free to modify the *items*, just not the *list*; so, as long as you do not modify the values of any item's next member, you are free to modify any other members.

In particular, this function *is* compatible with the pcb_queue_t and pcb_queue_node_t types.

Parameters

out	*item*	Iterator variable / loop index; will point to the current item (node) just before each execution of the loop body.
in	*list*	The singly- or doubly-linked list to iterate over.

Returns

Does *not* have a return value in the typical sense, however the value of the ouput parameter item is well-defined after the loop has terminated:

- If the loop terminates on its own, after iterating over the entire list, item will be NULL.
- Note that an empty list is a special case of the above, and in that case the value of item will be NULL after the loop has terminated, but the loop body will never have been executed.
- If you break out of the loop before it terminates on its own, item will point to the list item that was being processed during the iteration of the loop in

which `break` was called, *even if that item is the last item in the list.*

5.6.3 Enumeration Type Documentation

5.6.3.1 enum process_state_t

Type for variables that hold the state of a process.

```
             {

        RUNNING,
        READY,
        BLOCKED,
        SUSP_READY,
        SUSP_BLOCKED

} process_state_t;
```

5.6.3.2 enum process_class_t

Type for variables that hold the class of a process.

```
             {

        APPLICATION,
        SYSTEM

} process_class_t;
```

5.6.4 Function Documentation

5.6.4.1 void init_pcb_queues (void)

Must be called before using any other PCB or queue functions.

```
{
        queues[0] = &queue_ready;
        queue_ready.head                = NULL;
        queue_ready.tail                = NULL;
        queue_ready.length              = 0;
        queue_ready.sort_order          = PRIORITY;

        queues[1] = &queue_blocked;
        queue_blocked.head              = NULL;
        queue_blocked.tail              = NULL;
        queue_blocked.length            = 0;
        queue_blocked.sort_order        = FIFO;
        queues[2] = &queue_susp_ready;
        queue_susp_ready.head           = NULL;
        queue_susp_ready.tail           = NULL;
        queue_susp_ready.length         = 0;
        queue_susp_ready.sort_order     = FIFO;

        queues[3] = &queue_susp_blocked;
        queue_susp_blocked.head         = NULL;
        queue_susp_blocked.tail         = NULL;
        queue_susp_blocked.length       = 0;
        queue_susp_blocked.sort_order   = FIFO;
}
```

5.6.4.2 pcb_t∗ setup_pcb (char ∗ *name*, int *priority*, process_class_t *class*)

Creates, allocates, and initializes a new PCB object.

This function creates a new PCB object (pcb_t), then calls allocate_pcb() to do the allocation step. It then initializes the PCB's various fields according to both default values and the parameters passed in.

Returns

Returns a pointer to the new PCB, or NULL if an error occured.

Parameters

name	Name of the new process. Must be unique among all processes.
priority	Priority of the process. Must be between -127 and 128 (incl.)
class	Class of the process; one of APPLICATION or SYSTEM.

```
{
        /* Loop index. */
        int i;

        /* Pointer to the new PCB we're creating. */
        pcb_t *new_pcb;

        /* Check that arguments are valid. */
        if ( find_pcb(name) != NULL ) {
                /* Name is not unique. */
                return NULL;
        }
        if ( strlen(name) > MAX_ARG_LEN || name == NULL ) {
                /* Invalid name. */
                return NULL;
        }
        if ( priority < -127 || priority > 128 ) {
                /* Value of priority is out of range. */
                return NULL;
        }
        if ( class != APPLICATION && class != SYSTEM ) {
                /* Invalid class specified. */
                return NULL;
        }


        /* Allocate the new PCB. */
        new_pcb = allocate_pcb();
        if (new_pcb == NULL) {
                /* Allocation error. */
                return NULL;
        }


        /* Set the given values. */
        new_pcb->priority       = priority;
        new_pcb->class          = class;
        strcpy( new_pcb->name, name );


        /* Set other default values. */
        new_pcb->state          = READY;
        new_pcb->memory_size    = 0;
        new_pcb->load_address   = NULL;
        new_pcb->exec_address   = NULL;

        /* Initialize the stack to 0's. */
        for (i=0; i<STACK_SIZE; i++) {
                *(new_pcb->stack_base + i) = (unsigned char)0;
        }

        return new_pcb;
}
```

5.6.4.3 pcb_t∗ find_pcb (char ∗ *name*)

Finds a process.

Searches all process queues.

Returns

Returns a pointer to the PCB, or NULL if not found or error.

Todo

This really should be done a little cleaner, possibly using a foreach() macro, like the one at: http://stackoverflow.com/questions/400951/c-foreach-or-similar

Parameters

name	The name of the process to find.

```
{
        /* Pointer to the requested PCB, if we find it. */
        pcb_t *found_pcb;

        /* Validate arguments. */
        if ( name == NULL || strlen(name) > MAX_ARG_LEN ) {
                /* Invalid process name. */
                return NULL;
        }

        /* Search for the PCB.  If we find it, return it. */
        if ( found_pcb = find_pcb_in_queue( name, &queue_ready ) ) {
                return found_pcb;
        }
        if ( found_pcb = find_pcb_in_queue( name, &queue_blocked ) ) {
                return found_pcb;
        }
        if ( found_pcb = find_pcb_in_queue( name, &queue_susp_ready ) ) {
                return found_pcb;
        }
        if ( found_pcb = find_pcb_in_queue( name, &queue_susp_blocked ) ) {
                return found_pcb;
        }

        /* If we get here, the process was not found. */
        return NULL;
}
```

5.6.4.4 pcb_queue_t* insert_pcb (pcb_t * *pcb*)

Inserts a PCB into the appropriate queue.

Inspects the PCB's state member to determine which queue to insert into.

Inspects the queue's sort_order member to determine whether to insert in order of priority, or to simply insert the PCB at the end of of the queue.

Parameters

pcb	Pointer to the PCB to be enqueued.

```
{
        /* Pointer to the queue we will insert into. */
        pcb_queue_t                *queue;
        /* Pointer to the new queue node descriptor we must make. */
        pcb_queue_node_t           *new_queue_node;
        /* For use in loops that iterating through the queue. */
        pcb_queue_node_t           *iter_node;

        /* Validate argument */
        if (pcb == NULL) {
                /* PCB to insert cannot be null... come on :) */
                return NULL;
        }
        /* Determine which queue we will insert this PCB into. */
        switch (pcb->state) {
                case READY:
                        queue = &queue_ready;
                break;
                case BLOCKED:
                        queue = &queue_blocked;
                break;
                case SUSP_READY:
                        queue = &queue_susp_ready;
                break;
                case SUSP_BLOCKED:
                        queue = &queue_susp_blocked;
                break;
                default:
                        /* Unexpected value for PCB state (maybe Running?) */
                        return NULL;
                break;
        }

        /* Allocate the new queue descriptor. */
        new_queue_node =
                (pcb_queue_node_t *)sys_alloc_mem(sizeof(pcb_queue_node_t));
        if ( new_queue_node == NULL ){
                /* Error allocating memory. */
                return NULL;
        }


        /* Do the insert ... */
        /* ----------------- */

        new_queue_node->pcb = pcb;

        /* Case one: queue is empty. */
        if ( queue->length == 0 ){
                new_queue_node->next    = NULL;
                new_queue_node->prev    = NULL;
                queue->head             = new_queue_node;
                queue->tail             = new_queue_node;
                queue->length           = 1;
                return queue;
        }

        /* Case two: FIFO queue; we only need to insert at end. */
        if ( queue->sort_order == FIFO ){
                goto INSERT_AT_END;
        }

        /* The hard case: insert in priority-order. */
        iter_node = queue->head;
        while (iter_node != NULL) {
                if ( iter_node->pcb->priority < pcb->priority ){
                        /* Insert before iter_node */
                        new_queue_node->prev = iter_node->prev;
                        iter_node->prev->next = new_queue_node;
                        iter_node->prev = new_queue_node;
                        new_queue_node->next = iter_node;
                        if ( queue->head == iter_node ){
                                queue->head = new_queue_node;
                        }
                        queue->length++;
                        return queue;
                }
                iter_node = iter_node->next;
        }
        /* If we got this far, we need to do an insert-at-the-end. */


        INSERT_AT_END:
                new_queue_node->next    = NULL;
                new_queue_node->prev    = queue->tail;
                queue->tail->next       = new_queue_node;
                queue->tail             = new_queue_node;
                queue->length++;
                return queue;
}
```

Index