ECE 2400 Computer Systems Programming, Fall 2017

Prelim 1 Prep

revision: 2017-09-29-17-49

These problems are not meant to be exactly like the problems that will be on the prelim. These problems are instead meant to represent the kind of understanding you should be able to demonstrate on the prelim.

In the following problems, we will explore two different data structures with similar interfaces but very different implementations. These data structures could be used in an operating system to manage programs running on a server. Every program is given a unique *process ID*, and every program is also associated with the *username* of the user running the program. For example, the following commands will display your username and the process IDs of all programs you are current running on `ecelinux`:

```
% whoami
cb535
% ps
  PID TTY          TIME CMD
16734 pts/0    00:00:00 bash
17018 pts/0    00:00:00 ps
```

We will explore data structures to store the username and process ID for each program currently running on a system. The data structures should support queries by the operating system. The operating system should be able to query which user is running the program with a given process ID, and the operating system should also be able to list all process IDs associated with a given username. For this problem, you can assume process IDs range from zero to 31.

Problem 1. List-Based Program Info Data Structure

Our first data structure will be similar in spirit to the list data structure discussed in lecture. The data structure is named `pinfo_list`, since it is a list meant to store information about programs. We are using a list-based instead of a vector-based implementation, because we anticipate needing to frequently insert and remove entries as programs are started and then finish. The interface for `pinfo_list` is as follows.

```
typedef struct _entry_t
{
  int              pid;
  char*            uname;
  struct _entry_t* next_p;
}
entry_t;
```

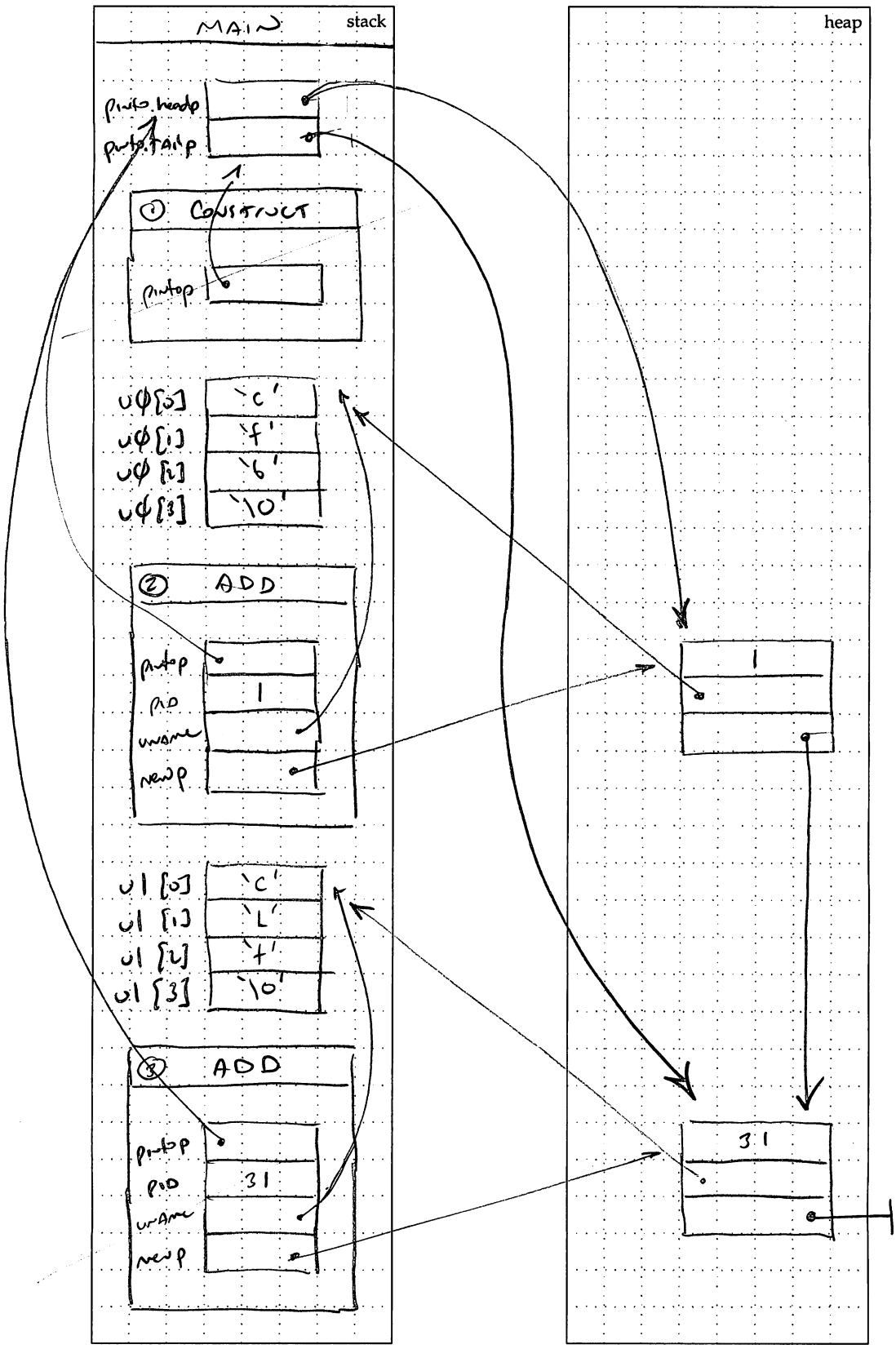
```
typedef struct
{
  entry_t* head_p;
  entry_t* tail_p;
}
pinfo_list_t;
```

```
void  pinfo_list_construct  ( pinfo_list_t* pinfo_p );
void  pinfo_list_destruct   ( pinfo_list_t* pinfo_p );
void  pinfo_list_add        ( pinfo_list_t* pinfo_p, int pid, char uname[] );
void  pinfo_list_remove     ( pinfo_list_t* pinfo_p, int pid );
char* pinfo_list_get_uname  ( pinfo_list_t* pinfo_p, int pid );
void  pinfo_list_print_pids ( pinfo_list_t* pinfo_p, char uname[] );
```

Part 1.A Implementing `pinfo_list_add`

The implementation for the `pinfo_list_construct` and `pinfo_list_add` functions are shown below. Notice that we always add the new entry to the end of the linked list. **Draw the stack frame and heap diagram that correspond to the execution of this C program.**

```
1  void pinfo_list_construct( pinfo_list_t* pinfo_p )
2  {
3    pinfo_p->head_p = NULL;
4    pinfo_p->tail_p = NULL;
5  }
6
7  void pinfo_list_add( pinfo_list_t* pinfo_p,
8                       int pid, char uname[] )
9  {
10   entry_t* new_entry_p
11     = malloc( sizeof(entry_t) );
12   new_entry_p->pid    = pid;
13   new_entry_p->uname  = uname;
14   new_entry_p->next_p = NULL;
15
16   // Handle case where list is empty
17   if ( pinfo_p->tail_p == NULL ) {
18     pinfo_p->head_p         = new_entry_p;
19     pinfo_p->tail_p         = new_entry_p;
20   }
21
22   // Handle case where list is not empty
23   else {
24     pinfo_p->tail_p->next_p = new_entry_p;
25     pinfo_p->tail_p         = new_entry_p;
26   }
27 }
28
29 int main( void )
30 {
31   pinfo_list_t pinfo;
32   pinfo_list_construct( &pinfo );
33
34   char u0[] = "cfb";
35   pinfo_list_add( &pinfo, 1, u0 );
36
37   char u1[] = "clt";
38   pinfo_list_add( &pinfo, 31, u1 );
39
40   return 0;
41 }
```


MAIN
stack
heap
plist.headp
plist.tailp
① CONSTRUCT
pintop
u0[0] 'c'
u0[1] 'f'
u0[2] '6'
u0[3] '\0'
② ADD
pintop
pID 1
uname
newp
1
u1[0] 'c'
u1[1] 'L'
u1[2] 'f'
u1[3] '\0'
③ ADD
pintop
pID 31
uname
newp
31

Part 1.B Implementing `pinfo_list_get_uname`

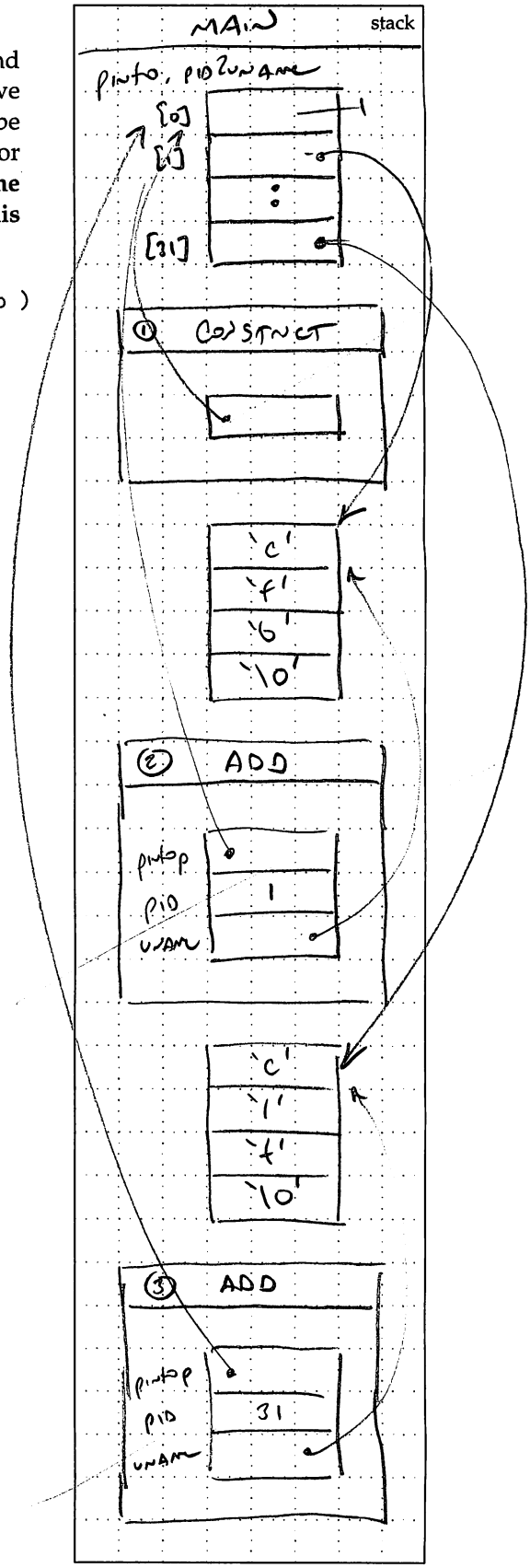
Develop an implementation for `pinfo_list_get_uname` function. The function should return the username corresponding to the given process ID (`pid`). If the `pid` is not present in the data structure, then the function should return a NULL pointer to indicate an error.

```
CHAR * PINFO_LIST_GET_UNAME ( pinfo_list_t * pinfo_p, INT PID)
{
  entry_t * entry_p = pinfo_p -> HEAD_p;

  while ( entry_p != NULL ) {
    if ( entry_p -> PID == PID )
      RETURN entry_p -> UNAME;
    entry_p = entry_p -> next_p;
  }

  RETURN NULL;
}
```

Part 2.A Implementing `pinfo_lut_add`

The implementation for the `pinfo_lut_construct` and `pinfo_lut_add` functions are shown below. Notice that we explicitly initialize all entries in the `pid2uname` array to be NULL. We will use a NULL pointer to indicate whether or not the corresponding entry is valid. **Draw the stack frame and heap diagram that correspond to the execution of this C program.**

```
1  void pinfo_lut_construct( pinfo_lut_t* pinfo_p )
2  {
3    for ( size_t i = 0; i < 32; i++ )
4      pinfo_p->pid2uname[i] = NULL;
5  }
6
7  void pinfo_lut_add( pinfo_lut_t* pinfo_p,
8                      int pid, char uname[] )
9  {
10   pinfo_p->pid2uname[pid] = uname;
11 }
12
13 int main( void )
14 {
15   pinfo_lut_t pinfo;
16   pinfo_lut_construct( &pinfo );①
17
18   char u0[] = "cfb";
19   pinfo_lut_add( &pinfo, 1, u0 );②
20
21   char u1[] = "clt";
22   pinfo_lut_add( &pinfo, 31, u1 );③
23
24   return 0;
25 }
```



Problem 2. Lookup-Table Program Info Data Structure

Our second data structure uses a new technique called a *lookup table*. A lookup table is an array where we can transform what we are searching for into an array index; then we can directly use the array index to access the corresponding entry. The data structure is named `pinfo_lut`; `lut` stands for lookup-table. The interface for `pinfo_lut` is as follows.

```
1  typedef struct
2  {
3    // This is an array of 32 char pointers
4    char* pid2uname[32];
5  }
6  pinfo_lut_t;
7
8  void  pinfo_lut_construct  ( pinfo_lut_t* pinfo_p );
9  void  pinfo_lut_destruct   ( pinfo_lut_t* pinfo_p );
10 void  pinfo_lut_add        ( pinfo_lut_t* pinfo_p, int pid, char uname[] );
11 void  pinfo_lut_remove     ( pinfo_lut_t* pinfo_p, int pid );
12 char* pinfo_lut_get_uname  ( pinfo_lut_t* pinfo_p, int pid );
13 void  pinfo_lut_print_pids ( pinfo_lut_t* pinfo_p, char uname[] );
```

Part 2.B Implementing `pinfo_lut_get_uname`

Develop an implementation for `pinfo_lut_get_uname` function. The function should return the username corresponding to the given process ID (`pid`). If the `pid` is not present in the data structure, then the function should return a NULL pointer to indicate an error.

```
CHAR * PINFO_LUT_GET_UNAME ( pinfo_lut_t * pinfo_p, INT PID)
{
    RETURN pinfo_p -> PID2UNAME [pid];
}
```

Part 2.C Implementing `pinfo_lut_print_pids`

Develop an implementation for `pinfo_lut_print_pids` function. The function should use `printf` to print every process ID corresponding to the given username (`uname`). You can use the `strcmp` function from the C standard library to compare to strings. Note that this function returns zero if the strings are equal.

```
VOID pinfo_LUT_PRINT_PIDS ( pinfo_LUT_t * pinfo_p, CHAR UNAME[])
{
    FOR (SIZE_t i =0; i< 32; i++) {
        IF ( STRCMP ( pinfo_p -> PID2UNAME [i], UNAME) ==0 )
            printf ("%d \n", i);
    }
}
```

Problem 3. Comparative Analysis

In this problem, you should perform a comparative analysis of the two implementations explored in Problems 1 and 2, with the ultimate goal of **making a compelling argument for which design will perform better across a large number of workloads.** While you are free to use whatever approach you like, we recommend you structure your response in several paragraphs. The first paragraph might discuss the performance of both implementations using time complexity analysis. Remember that time complexity analysis is not the entire story; it is just the starting point for performance analysis. The second paragraph might discuss the space requirements of both implementations using space complexity analysis. Remember that space complexity analysis is not the entire story; it is just the starting point for storage requirement analysis. The third paragraph might discuss other qualitative metrics such as generality, maintainability, and design complexity. The final paragraph can conclude by making a compelling argument for which implementation will perform better in the general case, or if you cannot strongly argue for either implementation explain why.