COL333
Asiignment-1

Sarthak
2020CS10379

February 2023

1 System Calls

1.1 *sys_print_count*

4 Functions namely "traceClear", "orderToSysCallNumber", "sysCallName" & "traceSysCall" and 3 arrays named "alphabeticalToSysCallNo", "sysNoToName", "counterForTrace" were defined in "syscall.c" having signatures present in "defs.h". Whenever the trace state is toggled from $TRACE\_OFF$ to $TRACE\_ON$, the function "traceClear" is called from inside of *sys_toggle* which resets all values in "counterForTrace" to 0. Upon any system successful system call, system call number was taken out from the %eax register and increment the "counterForTrace" of that particular system call. Whenever *print_count* is called, we iterate from 1 to 21, find the system call corresponding to alphabetical order(using the "orderToSysCallNumber" function), calculate the number of times it has been called since the last time toggle was switched on(using the "traceSysCall" function) and print it's name and number of times if it is > 0 using the *cprintf* function.

1.2 *sys_toggle*

An enum type called "traceState" was defined containing $\{TRACE\_ON, TRACE\_OFF\}$. A global instance of such a type called trace was initialized to $TRACE\_OFF$. Upon calling toggle, the state changes from $TRACE\_ON$ to $TRACE\_OFF$ and vice versa. Whenever the state of trace changes from $TRACE\_OFF$ to $TRACE\_ON$, the function "traceSysCall" is called which resets all values of "counterForTrace" to 0.

1.3 *sys_add*

This system call takes in 2 integers and returns their sum. The arguments were taken using the help of *argint* function defined in "syscall.c"(having signature in “defs.h”). And the sum of their values were returned if integer arguments were taken properly else (-1) was returned.

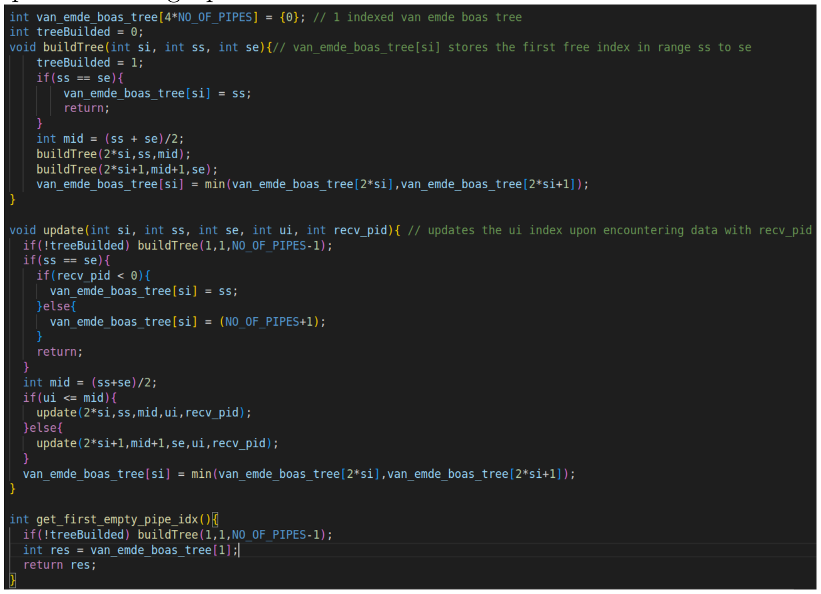
1.4 *sys_ps*

This system call prints a list of pids of all the current running processes along with their names. For this system call, a new function named “printAllRunningProcesses()” was defined in “proc.c”. This helper function, locks the ptable containing all the process structs, prints all those who do not have a *UNUSED* state and then releases the lock.

2 Inter Process Communication

2.1 Unicast

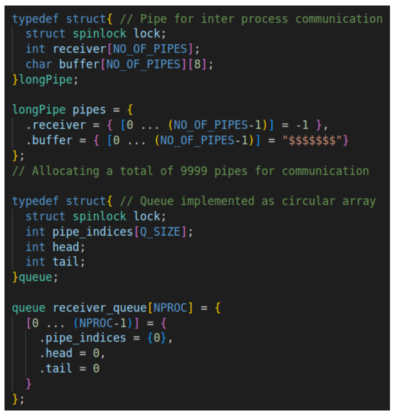
2 system calls namely *sys_send* and *sys_recv* were defined in this part. For this, a struct longPipe was defined as a large array(25000) of buffers. The Queue data structure was also implemented as a circular array by maintaining it’s head and tail with max length being the length of array. Such a queue was assigned for every process. An array of wait states are also initialized to *WAIT_OFF* indicating whether process with *recv_id* is waiting for a data or not. A variant of the Van-Emde-Boas-tree was also implemented on an array using ideas from segment trees. This allowed to find the first free index in longPipe in $O(1)$ time with updates in $O(logn)$ time. Code snippet for the tree and definitions of queue and longPipe :



```c
int van_emde_boas_tree[4*NO_OF_PIPES] = {0}; // 1 indexed van emde boas tree
int treeBuilded = 0;
void buildTree(int si, int ss, int se){// van_emde_boas_tree[si] stores the first free index in range ss to se
    treeBuilded = 1;
    if(ss == se){
        van_emde_boas_tree[si] = ss;
        return;
    }
    int mid = (ss + se)/2;
    buildTree(2*si,ss,mid);
    buildTree(2*si+1,mid+1,se);
    van_emde_boas_tree[si] = min(van_emde_boas_tree[2*si],van_emde_boas_tree[2*si+1]);
}

void update(int si, int ss, int se, int ui, int recv_pid){ // updates the ui index upon encountering data with recv_pid
  if(!treeBuilded) buildTree(1,1,NO_OF_PIPES-1);
  if(ss == se){
    if(recv_pid < 0){
      van_emde_boas_tree[si] = ss;
    }else{
      van_emde_boas_tree[si] = (NO_OF_PIPES+1);
    }
    return;
  }
  int mid = (ss+se)/2;
  if(ui <= mid){
    update(2*si,ss,mid,ui,recv_pid);
  }else{
    update(2*si+1,mid+1,se,ui,recv_pid);
  }
  van_emde_boas_tree[si] = min(van_emde_boas_tree[2*si],van_emde_boas_tree[2*si+1]);
}

int get_first_empty_pipe_idx(){
  if(!treeBuilded) buildTree(1,1,NO_OF_PIPES-1);
  int res = van_emde_boas_tree[1];
  return res;
}
```



```c
typedef struct{ // Pipe for inter process communication
  struct spinlock lock;
  int receiver[NO_OF_PIPES];
  char buffer[NO_OF_PIPES][8];
}longPipe;

longPipe pipes = {
  .receiver = { [0 ... (NO_OF_PIPES-1)] = -1 },
  .buffer = { [0 ... (NO_OF_PIPES-1)] = "$$$$$$$"}
};
// Allocating a total of 9999 pipes for communication

typedef struct{ // Queue implemented as circular array
  struct spinlock lock;
  int pipe_indices[Q_SIZE];
  int head;
  int tail;
}queue;

queue receiver_queue[NPROC] = {
  [0 ... (NPROC-1)] = {
    .pipe_indices = {0},
    .head = 0,
    .tail = 0
  }
};
```

Now we explain the implementation of *sys_send* and *sys_recv* in breif :

2.1.1 *sys_send*

This system call takes in the sender's process id, receiver's process id and the message to be sent, locks the pipes, uploads the message to the first index of free buffer in longPipe, and enqueues the index of buffer in the queue of receiver's process id, checks if the receiver is in wait state or not. If it is, then it toggles the wait state and wakes up all process sleeping at receivers end. Finally, it releases all the locks and returns 0 upon successful sending of data and -1 otherwise.

2.1.2 *sys_recv*

This system call takes in the buffer address to which a received data is to be stored. An infinite while loop is created which locks the pipes and the queue of the process calling itself, checks whether the queue has an element or not, if present, dequeues the front element, copies the data stored at longPipe index of the front element to the buffer address and releases the locks. If no element found in queue, sleep function is called on pipes, and the loop continues. Upon successful execution, 0 is returned, otherwise -1 is returned.

2.2 Multicast

In this section, the system call *sys_send_multi* was defined. This takes in a sender's process id, an array containing receiver's process id and a message to be sent. Note that it was assumed that the number of receiver's process id's is equal to 8. In order to implement the *sys_send_multi* function, we iterated over all the receiver's process id array, locked the resources and sent the data to them individually, releasing the resources at the end of every iteration.

3 Distributed Algorithm

In this part of the assignment, A multi-processor program to sum the elements of an array and to calculate it's variance was to be designed with the conditions that the total number of elements in the array = 1000 and each element ranges between 0 and 9. The code for this part is done in the assign1_8.c file. The program takes in 2 command line arguments, the first one being an integer type and second one being the input file name containing the array. Type being 0 denotes a unicast based approach while type = 1 denotes a multicast approach. We describe the following approaches below :

3.1 Unicast Approach

The main function is forked 8 times and each child process is given the task to calculate the partial sum of array from $125 \times i$ to $125 \times (i+1)$ and send the partial sum to the main process(by calling the $sys\_send(getpid(), main\_pid, (void*)\&partialSum)$. After which, the main process receives the partial sums from it's queue and sums them to the *tot_sum* to get the aggregate sum of the array.

3.2 Multicast Approach

In this approach, the total sum is evaluated as in Unicast algorithm. The main process then divides the sum with 1000 in floating point arithmetic to get the average of the array. Again fork is called 8 times on the main process, their id's are stored in an array *child_ids* and each process waits for data from the main process. The average is sent to the children processes using the *sys_send_multi* function, after which each process evaluates $\sum\limits_{125i}^{125(i+1)-1} |a_j - avg|^2$, sends this value to the main process and exits. The main process waits for all children to send their calculations and exit, after which it sums them and divides by 1000 to yield the variance.
The variance is finally printed using the *print_variance* function given on piazza.