OPERATING SYSTEMS: ASSIGNMENT 1

Shivam Verma (2020CS50442)

1. SYSTEM CALLS

- **TRACE**

  For this system call, a global array is maintained which is initially initialised to 0 and the indices correspond to the different system calls. Inside the `syscall` function in `syscall.c` file, whenever a valid system call is made, the value at that particular index is incremented in the array if the current state is `TRACE_ON`.

  Since `SYS_toggle` and `SYS_print_count` are not required to be printed, they are ignored.

```
if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    curproc->tf->eax = syscalls[num]();
    if (current_trace_state == TRACE_ON && num != SYS_toggle
&& num != SYS_print_count) {
      system_call_count[num]++;
    }
}
```

- **TOGGLE**

  It simply flips the `current_trace_state` from `TRACE_ON` to `TRACE_OFF` and vice-versa. Also the count array is reset to 0, every time the toggle is called.

```
if (current_trace_state == TRACE_ON) {
    current_trace_state = TRACE_OFF;
} else {
    current_trace_state = TRACE_ON;
}
for (int i = 0; i < TOT; i++) {
    system_call_count[i] = 0;
}
```

- **ADD**

  The parameters passed to this function are extracted from `argint` functions, since system calls in `xv6` have `void` as argument. So the two operands are extracted and their sum is returned.

```
int a, b;
int i = argint(0, &a);
int j = argint(1, &b);

if (i < 0 || j < 0) {
  return -1;
}
return a + b;
```

- **PROCESS LIST**

  To print the list of all current running processes, the process table is iterated and for all processes whose state is not `unused`, they are printed.

```
struct proc *p;
acquire(&ptable.lock);
  for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
    if (p->state == UNUSED)
      continue;
    cprintf("pid:%d name:%s\n", p->pid, p->name);
  }
release(&ptable.lock);
```

2. INTER PROCESS COMMUNICATION

For implementing IPC, some additional structures are defined. A `message` struct is defined which stores the 8-byte message, location in the buffer and the pointer to the next message struct. Also for each process there will be a `msgQueue`, which will be the queue of messages each process has received from other processes.

So whenever process A has to send a message to process B, it will find an empty buffer location, and copy its message content in the buffer. Then the message from the buffer will be enqueued to the receiver's `msgQueue`.

- **WHY IS INTERMEDIATE BUFFER USED**

  The message is temporarily stored in the buffer because the message passing system needs to allocate memory for the incoming message before adding it to the recipient's message queue.

  By using a buffer, the message passing system can quickly allocate memory for incoming messages without having to perform a slow memory allocation operation for each message. The buffer also allows for better control and management of memory allocation, as the buffer can be implemented with a fixed size or with a dynamic allocation strategy.

  Additionally, the buffer can be used to temporarily store messages if the recipient's message queue is full, allowing the sender to continue to send messages without being blocked.

  If inter process communication was implemented without using buffers, then the message would have to be sent directly from the sender process to the receiver process. This would mean that the message would have to be copied from the sender's address space to the receiver's address space, which would require a complex memory management mechanism to ensure that the message is delivered correctly.

  Without a buffer, there would be no temporary storage for the message, and the message would have to be transferred immediately from the sender to the receiver. This would result in a lot of overhead, especially if the message is large or if there are many messages being sent at the same time. Additionally, without a buffer, the sender process would have to wait until the receiver process is ready to receive the message, which could result in blocking and reduce the overall efficiency of the system.

- **SPINLOCKS ARE USED IN SENDING AND RECEIVING FUNCTIONS**

In the `sys_send` function, spinlocks are used to protect shared data structures and ensure synchronization between the sender and receiver processes. The spinlocks are used to enforce mutual exclusion between the sender process and any other processes that might be accessing the same data structures.

The spinlocks are used to prevent the sender process from modifying the message queue while other processes are accessing it. This helps prevent race conditions, where two or more processes might try to modify the same data structure simultaneously, leading to incorrect results.

For example, in the `sys_send` function, the spinlock `"msgQueue_locks"` is used to protect the message queue `"QQ"`. Before the sender process inserts a message into the queue, it acquires the spinlock `"msgQueue_locks"`. This ensures that no other process can access the message queue until the sender process has finished inserting the message. Once the message has been inserted, the sender process releases the spinlock `"msgQueue_locks"`, allowing other processes to access the message queue again.

A Message Queue Data Structure is defined for storing messages for each process and bookkeeping the transactions. Here is a brief description of what different functions do:

- `setup`: This function initializes the message queue by setting the front and last pointers to 0, indicating that the queue is empty.
- `add_in_queue`: This function adds a new message to the end of the queue. The `n` argument is a pointer to the message to be added, and `Q` is a pointer to the message queue. If the queue is empty, the front and last pointers are set to `n`. Otherwise, `n` is added to the end of the queue, and the last pointer is updated to point to the new last message.
- `remove_from_queue`: This function removes the first message from the queue and returns a pointer to it. If the queue is empty, the function returns a null pointer. If the queue is not empty, the front pointer is updated to point to the next message in the queue, and the last pointer is updated if the queue becomes empty as a result of the removal.

Together, these functions allow you to create and manipulate a message queue data structure that can be used for interprocess communication.

UNICAST MODEL

Process sends message to another process using `sys_send` system call, and receives message using `sys_recv` system call. Here is a simple flowchart of how `Unicast Model` will look like:

1. A sender process invokes the `sys_send` system call, passing the message to be sent and the identifier of the receiver process.
2. The `sys_send` function takes the message and stores it temporarily in a buffer, such as the `msg_buffer` array.
3. The function finds a free buffer position in the `msg_buffer` array and sets it as allocated.
4. The function creates a message object, `final`, by pointing to the buffer position, filling it with the message data and the buffer position, and setting its next pointer to null.
5. The function acquires the lock for the message queue of the receiver process, adds the message object to the queue, and releases the lock.
6. The function sends a signal to the receiver process to indicate that a message is waiting in its queue.
7. The receiver process invokes the `sys_recv` system call to receive a message.
8. The `sys_recv` function acquires the lock for its own message queue and removes the message object at the head of the queue.
9. If the message queue is empty, the function blocks until a message is added to the queue.
10. The function releases the lock and copies the message data from the message object to the buffer passed as an argument to the `sys_recv` call.
11. The function updates the buffer allocation status and returns to the receiver process.

MULTICAST MODEL

The `Multicast Model` is implemented without Interrupt Handlers. So the same `sys_recv` system call is used by a receiver's process to receive a message. Same protocol as above is followed. In case of sending message to multiple processes simultaneously, the sender process iterates through the `rec_pids` array and then sends the message to each valid process using the method described above.

Also the size of `rec_pids` array is set to 8. This means a multicast model can involve at-max 8 processes. Whenever there are less than 8 processes, rest of the pids are set to `-1`. These illegal pids are ignored by the sender process while multicasting.

This is the basic code snippet of sending function, where first a `buffer_lock` is acquired. The buffer is then iterated to find an empty location, where the message is temporarily transferred. Finally a `msgQueue` lock is acquired concerning the receiving process and the message is added to its queue. The receiving process is `unblocked` if it was in a `blocked` state waiting for a message to arrive:

```
acquire(&buffer_lock);
for(int i=0; i < NELEM(msg_buffer); i++) {
  if(allocated[i] == 0) {
    allocated[i] = 1;
    release(&buffer_lock);
    final = &msg_buffer[i];

    final->bp = i;
    memmove(final->msg,msg,8);
    final->next = 0;

    acquire(&msgQueue_locks[rec_pid]);
    add_in_queue(final, &QQ[rec_pid]);  // inserted in queue
    unblock(rec_pid);
    release(&msgQueue_locks[rec_pid]);
    return 0;
  }
}
```

- **The sys_recv function is a blocking call**

  The sys_recv function is a blocking call because it waits for a message to be received by the receiver process before it continues execution. This means that the receiver process will be blocked, or suspended, until a message is received. The implementation of sys_recv in the code uses a spinlock to control access to the message queue and ensure that only one process can receive a message at a time. The function checks if the message queue is empty, and if so, it waits until a message is placed into the queue by another process. This is achieved by repeatedly checking the status of the queue and waiting for a change, using a spinlock to control access to the queue. Once a

message is available, the sys_recv function retrieves the message from the queue, removes it from the queue, and returns it to the caller. By waiting for a message to be received, the sys_recv function ensures that the receiver process does not continue executing until it has a message to process, making it a blocking call.

3. DISTRIBUTED ALGORITHM

Following is the detailed explanation of `DISTRIBUTED ALGORITHM` which computes `SUM` and `VARIANCE` of elements of an array in parallel and distributed fashion.

- There is a coordinator process, whose job is collect partial data from forked processes and do the computation. The process id (PID) of this master process is stored in `coordinator` variable.
- The number of child processes should be between 1 and 8. They are created using the `fork()` command. The `rec_pids` array stores all the PID's of these child process. It will be later used in `multicast` communication.
- The `divide` variable stores the chunk of area each child process will work upon. It ensures the distribution of computation among various processes.
- Each child process calculates the sum of its chunk, and send this partial sum to the coordinator using `send(getpid(), coordinator, &partial_sum)` command. In case of `type == 0`, the children processes exit at this stage, as no further work is required from them.
- The coordinator process upon receiving the partial sums, adds them together to compute the total sum of array and its mean. In case of `type == 1`, the coordinator multicasts this mean to all the children processes using `send_multi(coordinator, rec_pids, &temp)` command.
- The child processes receive the mean and calculate the `sum of square of difference about the mean` in their respective chunk and send this partial data back to coordinator using `unicast`.
- The coordinator finally collects these partial sum of square, add them up and divide by 1000, to obtain the variance.

  PARENT PROCESS RECEIVING PARTIAL SUMS FROM CHILD PROCESSES AND MULTICASTING MEAN TO ALL IN CASE OF TYPE 2:

```
1  int aux = 0;
```

```
int S = 0;
while (aux < child_processes) {
  int *msg = (int *)malloc(8);
  int stat = -1;
  while (stat == -1) {
    stat = recv(msg);
  }

  S += *msg;
  ++aux;
}

tot_sum = S;
if (type == 1) {
  mean = tot_sum / (1.0 * size);
  float temp = mean;
  send_multi(coordinator, rec_pids, &temp);
}
```

CHILD PROCESSES RECEIVING MULTICASTS, CALCULATING PARTIAL VARIANCE AND SENDING BACK TO COORDINATOR:

```
if (pid == 0) {  // child msg receiving multicasts
  while (mean == -1) {
    float *msg = (float *)malloc(8);
    int stat = -1;
    while (stat == -1) {
      stat = recv(msg);
    }
    mean = *msg;

    if (mean >= 0) break;
  }
  float partial_var = 0.0;

  for (int i = tid * divide; i < min(size, (1 + tid) * divide);
i++) {
    partial_var += (mean - arr[i]) * (mean - arr[i]);
  }
  send(getpid(), coordinator, &partial_var);
  exit();
  // work done, now exit it
}
```