IMPLEMENTATION DETAILS - AYUSH JAIN [jain207@purdue.edu]

Specific questions asked in lab5 for implementation are at the end

Steps taken for the implementation

Process Cleanup Function

a) The pointer to the process cleanup function (provided by the user) is stored in the process entry in proctab corresponding to each process.
b) Whenever regcleanup system call is called with the function pointer to process cleanup function, the pointer in the corresponding process entry for the current process is set to the user provided pointer.
c) If NULL is sent as an argument, the corresponding function pointer is set to NULL without any error.
d) The cleanup function (if not NULL for a particular process) is called in kill system call to perform additional cleanup(like buffers, bufferpool etc.) apart from the usual stack deallocation in kill as per functionality provided by the user in the process cleanup function.
e) This works when kill is called for a particular process either explicitly or implicitly through process termination inside userret function.

Retrieving buffer pool information and implementation of freebufpool

a) In order to retrieve information about number of buffers allocated for a particular buffer, we use the semaphore count/value of bpsem in the struct bpentry for that corresponding buffer pool in buffer pool table *buftab*.
b) This semaphore value indicates the current available buffers in the buffer pool.
c) Also, the total number of available buffers allocated for a buffer pool in mkbufpool system call are fixed. As a result, a bufcount variable is added in the struct bpentry to keep count of total number of buffers that can be allocated for a particular buffer pool. This value is added to the totalbufs argument of bufsavail system call.
d) Both the available buffers and totalbufs count are checked for equality in freebufpool system call. If equal, then bufferpool allocated by a particular process is deallocated else system error is returned. Also, corresponding buffer pool entry in the buftab along with the allocated available buffers for that pool are deallocated so that the entry can be reused by any other new buffer pool.
e) Also, the poolid is retrieved for a new bufferpool in mkbufpool systemcall by finding the first available buffer pool entry in buftab in a similar fashion of newpid() corresponding to proctab. It is checked whether the bpnext pointer is null and no semaphore is allocated for an empty available buffer pool.
f) It is assumed that freebufpool will be called as a part of process cleanup function provided by the user **after** deallocating all it's allocated buffer across different buffer pools. This will ensure that bufferpool has a chance of being empty after deallocation of buffers by a process if pool's buffers are not shared by any other process. If shared, then it returns SYSERR.

System vs User processes

In order to differentiates between the 2 types, a snapshot(pids) of all the system processes are stored right before *main* user process creation in bool8 sys_proc array of size NPROC. This information is used in restart system call when only user processes need to be restarted.
This info is stored in record_system_processes() function in initialize.c just before main process creation.

Restart System Call

a) In order to kill and restart all the user processes, a new user process is created which calls kill on other user processes. If a process has registered a process cleanup function, then it is automatically called in kill for that process performing additional cleanup.
b) Sleep is called introducing a delay after killing all user processes as provided by the argument in restart system call.
c) For restarting the all the user process, snapshot of all the arguments is used corresponding to each process when the processes are created in create system call.
This snapshot is of following type struct in memory.h :

```
struct createinfo {
        void      *funcaddr;        /* Address of the function          */
        uint32    ssize;            /* Stack size in bytes              */
        pri16     priority; /* Process priority > 0             */
        char      *name;            /* Name (for debugging)             */
        int32     nargs;            /* Number of args that follow       */
        uint32  *list;
        pid32     oldpid;           // pid of the user process that is recreated
};
```

The struct array of NPROC size is created in initialize.c as *infotab*

The process is recreated using the old pid of the user process to be restarted and the corresponding entry for that oldpid containing arguments info, stack size etc. Since the number of arguments can be dynamic, a pointer is stored as list which is dynamically allocated and freed based on the number of arguments in create system call.
Also, the sys_proc info is used at this execution point to kill and recreate only user processes.

EXTRA CREDIT

For implementation of extra credit requirement, the memory allocated on the heap by system processes is recorded by storing the snapshot of the freelist implementation by allocating memory based on the number of blocks in the freelist. Count * 8 bytes memory is allocated to store all the info about freelist before main creation in the method freelist_snapshot() in initialize.c

The restore_freelist function is called in the restart function call right before delay to restore the freelist to the stored snapshot. The function has been commented as their is slight discrepancy (for few memblks in freelist) in the restored snapshot of freelist. Although, the user processes are being restarted without any issue, the function has been commented as potential small issues in restoration can disrupt the recreation of user processes.

Extra Credit Question

Question: How are buffer pools handled as part of the freelist restore from the snapshot?

In the current implementation, it is assumed that user will take care of freeing buffers and bufferpool as they have been allocated by the user and it is the user's responsibility to deallocated them when they have achieved their purpose.

Advantages of this approach

i) Less complexity on the OS side to keep track of all the allocation of buffer and bufferpools.

ii) The process cleanup function automatically called in kill system call if present. Adding code for freeing buffers and buffer pools is highly convenient to user as per OS design.

Disadvantages

i) Improper process cleanup function provided by the user can lead to memory wastage which can accumulate over multiple restart calls. This can lead to system crash if not handled by OS.

Extra Credit Study Question

Does this behavior of restart have any practical use?

- **Is there a better alternative behavior of restart?**
- **What are the advantages and disadvantages of either implementation?**

Consider how processes can communicate in Xinu: Does restarting all user processes create any timing problems that might affect inter processes communication?

Disadvantages

Calling restart system call in the main user process (where main is the source of all other user processes) without any global restart flag does not have any practical use as it spawns duplicate user processes by calling restart again and again. This not only waste CPU resources but can be incorrect as the functionality of process can be such that they are supposed to run only once.

Example Scenario

Consider a counting semaphore created by the main process and other user processes created by the main waiting on that semaphore. Creating duplicate user processes can lead to a situation of deadlock as additional wait calls can be made by the duplicate processes. Thus, it can affect the inter-process communication in a very negative manner by creating timing problems.

Advantages

Only advantage of this approach without any global flag is that no user process is responsible to keep track of restart system calls by using global restart flag. This reduces the complexity of multiple restart.

But, having a global flag restricts the restart to original user processes only which is the usual desired behaviour in most scenarios.

<u>Specific Questions asked in the Lab</u>

i)What specific problems can occur if user processes are repeatedly killed and restarted?
ii) Suppose a set of user processes is killed in random order. Show how the system can be left in a state where some processes never die?

i) If use processes are repeatedly killed and restarted, then duplicate processes can be created if global flag is not used (as discussed above). Also, incorrect deallocation of buffers and buffer pools in process cleanup functions during restart process can lead to accumulation of unused memory leading to memory wastage.

ii) Some processes may never die because it is possible that restart processes are called continuously by user processes in the middle of a current restart execution. This is assuming that global flag is not used. This can happen if a context switch happens during the delay phase of restart system call and that leads to execution of another restart system call causing user process to never terminate.

Test Cases are organised by testing only [freebufpool+availbufs+regcleanup] + [restart sys call] and then all of them combined along with existing functionality system calls like mkbufpool, getbuf, freebuf. More details for each test case in comments in main.c for each specific test case.