3

To implement execution of a callback function when a process receives a message, two entries are added to the process table:

```
int (* msg_cb_func) (void);
char load_msg_callback;
```

During a regcallback() call, the load_callback flag is set to 1 and `msg_cb_func` is set to the passed function.

Due to the single processor design, the process that registered a callback will have to be context switched out or give up cpu to allow a different process to send a message. That is, process will resume again from a reschd() call. Therefore, within the reschd() routine after the ctxsw() call, the kernel can check if a callback needs to be run by checking the prhasmsg flag. This assures the callback is executed in the context of the process that registered the callback function.

The `load_msg_callback` flag is used as a safety check to verify a callback function is set before a callback is called once a message is detected in the message buffer.

The callback function used for testing is as follows:

```
int32 msg_recv_cb(void)
{
    umsg32 msgbuf;
    msgbuf = receive();
    kprintf("[%d] callback executed.\nmsg: %u\n", getpid(), msgbuf);
    return(OK);
}
```

All tests can be found under system/testCallback.c.

Test 1 - Single Sender + Single send() and Single Recipient

A recipient registers a callback, goes to sleep and enters and indefinite loop to make sure it is still running once the sender is done sending. A sender is created and it sends the message to the sleeping recipient.

Output:

Format: **[**<*current process id*>**] …**

```
Testing Callbacks

Test 1
[3] callback set!
[4] sent 100 to 3
[3] Calling callback at address 1062629
[3] callback executed.
msg: 100
```

As observed, the callback function is executed in the context (privilege level) of the process that registered the callback.

Test 2 - Single Sender + Multiple send() and Single Recipient

The setup is same as above but the sender executes multiple sends during the recipient process's execution. Due to the design of send() the recipient should get a different message everytime a callback above is executed.

Output:

```
Test 2
[3] callback set!
[4] sent 100 to 3
[3] Calling callback at address 1062642
[3] callback executed.
msg: 100
[4] sent 200 to 3
[3] Calling callback at address 1062642
[3] callback executed.
msg: 200
[4] sent 300 to 3
[3] Calling callback at address 1062642
[3] callback executed.
msg: 300
```

As expected, the callback function is executed for every send call until the recipient process is killed or sender stops sending

NOTE: Handout does not specify whether the callback function once set should be repeated for every send. This design executed the same callback for every send call. The design can be easily changed by setting the load_callback flag to 0 after executing the callback for the first time and therefore requiring subsequent regcallback() calls to make the callback execute similarly again.

Test 3 - Multiple Senders + Multiple send() and Single Recipient

Output:

```
Test 3
[3] callback set!
[4] sent 900 to 3
[3] Calling callback at address 1062642
[3] callback executed.
msg: 900
[5] sent 100 to 3
[3] Calling callback at address 1062642
[3] callback executed.
msg: 100
[4] sent 800 to 3
[3] Calling callback at address 1062642
[3] callback executed.
msg: 800
[5] sent 200 to 3
[3] Calling callback at address 1062642
[3] callback executed.
msg: 200
[3] killed
```

The output shows that the callback function is executed for every send the a sender process executes.

4

To implement a signal handling subsystem, the following fields were added to the process table entry:

```
pid32 child_pr_killed; // set to 1 when kill is called on a child
char monitor_child; // used for XINUSIGCHLD
uint32 pr_start; // set to clktimefine during create
char load_msg_callback; // flag to check if valid call back is set
uint32 wall_time_limit;  // set during system call to register callback to wall time
char wall_time_set; // flag to check if valid wall time set

int (* msg_cb_func) (void); // set to the address of callback func during regcallback
int (* wall_cb_func) (void); // set to the address of callback func during regcallback
int (* chld_cb_func) (void); // set to the address of callback func during regcallback
```

The comments state the functionality of each field. The last three fields store the callback function that corresponds to each signal.

The major difference in the design is to check for the wall time crossing even when a process is the only process running since this is the only signal that can be generated without involving other processes. Therefore, before reschd() continues the same process due to the process's highest priority, the wall_time_set flag is checked.

NOTE: The handout does not specify if the wall time callback should execute every time a process resumes execution after crossing the wall time. This design executes the corresponding callback once and sets the wall_time_set flag to 0.

Another difference in the design is to set a flag in kill() method to notify a process's parent process when a child process terminates.

Test 1 - Wall time crossing single process

The following callback function is used to check if wall time is crossed:

```
int32 wTimeStatus(void)
{
    kprintf("[%d] Crossed wall time @ clktimefine: %u\n", getpid(), clktimefine );
    return OK;
}
```

See c_test1() in testSignal.c
Output:

```
Test 1
[2] resuming process with pid: 3 @ clktimefine:
[3] Crossed wall time @ clktimefine: 5013
[2] treminated proc: 3 @ clktimefine: 20013
```

Test 2 - Message signal with XINUSIGRCV

The main method sends a message to the recipient process after the recipient registered the callback. Output:

```
***     Testing signal handlers         ***

[2] Sending 98 to proc: 3
[3] callback executed.
msg: 98
```

The test cases from **3** above also produce the expected results.

Test 3 - XINUSIGCHLD + multiple children terminated

The following callback is called when a child terminates:

```
int32 chldTerm(void)
{
    kprintf("[%d] callback executed. A child process was terminated\n", getpid() );
    return OK;
}
```

A new process *p1* is created that creates it's own 3 child processes after registering a callback. Later, the main process kills *p1*'s child processes.

Output:

```
Test3
[3] creating child processes
[4] process created
[5] process created
[6] process created
[2] terminating child processes: 4, 5, 6
[3] callback executed. A child process was terminated
[3] callback executed. A child process was terminated
[3] callback executed. A child process was terminated
```

As the output shows, the process that registered the callback and created the child processes, executes the callback every time a child process ends. Note that the children have to be terminated by main with sleep calls in between the kill() calls. Otherwise the callback will only execute once since every killl() call sets a single flag in the parent's process table entry.

Test 4 - Concurrent Client/Server App + waitchld()

Two processes are created. A "server" and a client. The server first creates a child process called init(). Once init() terminates (known through waitchld()), the server process waits for messages in the buffer by registering a callback with XINUSIGRCV.

Once it finishes serving the client, it creates a process called cleanup(). The server uses waitchld() to wait till clean up exits and prints the final terminating message. cleanup() sleeps for 5 seconds. See test5() to see more details in *system/testSignal.c*.

The print statements are affected by context switching. The key point is to see that waitchild() is successfully used to verify a specific child process created by a parent has exitited. This is tested with the following if statement:

```
kprintf("[%d] server completed serving clients. @ clktimefine: %u\n", getpid(), clktimefine );
pid32 cl_id = create(cleanup_server, 2000, 20, "server_cleanup", 0 );
resume(cl_id);
dc = waitchld();
if(dc == cl_id)
    kprintf("[%d] cleanup proc exited. Exiting server @ clktimefine: %u\n", getpid(), clktimefine);
```

Output:

```
[3] server completed serving clients. @ clktimefine: 10262
[cleanup] cleaning up server
[3] Client sent msg: 6
[3] Sending response: -1 to client
[3] cleanup proc exited. Exiting server @ clktimefine: 15270
```

Test 5 - All 3 callbacks set and corresponding 3 signals are raised

When a different callback function is set for each of the signals XINUSIGRCV, XINUSIGXTIME and XINUSIGCHLD. When all three events are triggered, 3 callback functions will be executed.

The message received by process (pid:) 3 is the message the terminating child sends to the parent before kill finishes execution on the child.

Bonus

To implement garbage collection, a new data structure (`struct allocated_mem`) was created. It can be found in `include/garbage_collection.h`. The process table entry also gets a new field `dmem` that is used to keep track of the block addresses and the the size of allocated blocks given out by getmem(). When a new process is created, the `dmem` field is initialized to an empty set in create(). When a process calls getmem(), the returned block and its block size are saved in `dmem`. If a process calls freemem(), the freed block is removed from `dmem`.
Once a process ends, the execution is turned over to kill(). Within kill(), any block remaining in `dmem` is freed before the execution of kill can continue.

Testing

Uncommenting the print statements around the untrackBlock() and trackBlock() calls in getmem() and freemem(), we can see that when a process uses getmem() the kernel prints:
Format: *[func][pid] ...*

```
[getmem][3] tracking block. blkaddr: 2235760
dmem : { 2212320 2235760 }
```

Testing the garbage collection is most effective using print statements. A process allocates memory and prints the address:

```
char * f = getmem(23435);
kprintf("[%d] got block %u\n", getpid(), f);

char * g = getmem(999);
kprintf("[%d] got block %u\n", getpid(), g);

freemem(f, 23435);

char * h = getmem(2222222);
kprintf("[%d] got block %u\n", getpid(), h);
```

The blocks are tracked and untracked as shown by printing `dmem` between the getmem() and freemem() calls:

```
[getmem][3] tracking block. blkaddr: 2212320
dmem : { 2212320 }
[3] got block 2212320
[getmem][3] tracking block. blkaddr: 2235760
dmem : { 2212320 2235760 }
[3] got block 2235760
dmem : { 2212320 2235760 }
[freemem][3] untracking block. blkaddr: 2212320
dmem : { 2235760 }
[getmem][3] tracking block. blkaddr: 2236760
dmem : { 2235760 2236760 }
[3] got block 2236760
```

freemem() is never called on `g` and `h`. Later when the process is terminated, freemem() is called on the unfreed blocks as evident by the output from kill() below.

Krutarth Rao
(0027262283)