Checkpoint 4 Writeup

Did we make the checkpoint? YES!! WAHOO!!

Please let us know if we can make anything clearer for the next checkpoint.

What We Did

In `syscalls.c` we have the code for `Fork`, `Wait`, `Delay`, `Exec` etc.

We've done lots of testing, check the testing section below!

How did we implement round robin?

- by using a queue to represent ready processes, when the ClockInterrupt is triggered, we get the process at the front of the ready queue, and move the previously-active process to the back of the queue

Implementation Changes

We're no longer using a bit_vector to keep track of free frames but a linked list! The functions related to it are in `list.c` and `list.h`, instead of a global `bit_vector` being defined in `kernel.c`, a `pfn_list` is declared instead.

Also, we've separated code for syscalls and the interrupt vector table to avoid scrolling. Our Kernel syscalls are in `syscalls.c`, but the interrupt vector table and the CASE statements that call these functions are still in `traphandlers.c`.

Testing

In this section, we go through how we tested each syscall.

Delay - `init.c`

`progs/init.c` tests Delay, so we test it by calling `./yalnix progs/init -W`

`init.c` traceprints something, delays for 1 clock tick, traceprints something else, delays for 2 clock ticks, traceprints, and repeats that. The checking for this one is more looking at whether or not our Kernel stays in idle for the correct number of clock ticks, because of that we also traceprint a Clock Tick that starts at 0 at the start of kernel boot, and increments for each Clock Tick.

Here's the code to save you from jumping around windows:

```
int main(int argc, char const *argv[]) {
    int pid = GetPid();
    while (1) {
        TracePrintf(1, "init.c: PID -> %d\n", pid);
        Delay(1);
        TracePrintf(1,"init.c: delay 2: PID -> %d\n", pid);
        Delay(2);
        TracePrintf(1,"init.c: delay 3: PID -> %d\n", pid);
        Pause();
    }
    return 0;
}
```

- tracing for this is in `test_traces/DELAY_TRACE`
- there's quite a bit of scrolling in the TRACE file through all the initialization information but once init.c (or pid 1) exits from KernelStart, by standard it Brks, and GetPids, then it prints `User Prog  init.c: PID -> 1      PPID -> 0`, and then we can see from Hardware that there's a syscall trap Delay. Our kernel also says `Yalnix      kernel calling Delay(1)` We switch to idle, and then switch back at the next clock tick (Clock Tick 2), we switch back to `init.c`, our program prints `User Prog  init.c: delay 2: PID -> 1     PPID -> 0`, then we see a syscall trap delay from hardware, and again, our kernel says `Yalnix      kernel calling Delay(2)`. We see while in our idle program, with each clock tick, that there is a process in the blocked queue (`Yalnix      Clock Tick -> 3 Yalnix      Ready -> 0 ::: Blocked -> 1 ::: Defunct -> 0`) Then we switch back at Clock Tick -> 5 which makes sense.

Brk - `brk.c`

`progs/brk.c` tests Brk, so we test it by calling `./yalnix progs/brk -W`

`brk.c` keeps mallocing massive chunks of memory in an eternal loop, it'll print whether or not the result of the malloc is NULL. Other print statements from within the kernel code will traceprint information about allocating pages and behavior expected of Brk.

Here's the code to save you from jumping around windows:

```
int main(int argc, char const *argv[]) {

    int pid = GetPid(); // get pid and print
    TracePrintf(1,"brk.c: PID -> %d\n",pid);

    int malloc_size = 100000;

    while (1) { // keep mallocing!!!
        TracePrintf(1,"brk.c: going to keep mallocing %d...\n",malloc_size);

        void* big = malloc(malloc_size);

        if (big == NULL) { // if malloc fails!
            TracePrintf(1,"brk.c: malloc returned NULL!\n");
        }
        else { // if malloc doesn't fail!
            TracePrintf(1,"brk.c: malloc successful, it's at %p\n",big);
        }
    }
}
```

- tracing for this is in `test_traces/BRK_TRACE`
- *lazy checking*: if you do `cat test_traces/BRK_TRACE | grep brk.c | less` it'll show all the traceprint statements from `brk.c`, I recommend including `less` or `head` because we let the program run on for a bit too long. But you can see that initially, we're successfully mallocing memory, the address of malloc'd memory is moving up, and after a while we run out of memory and malloc returns NULL. This shows that brk is failing gracefully!
- *not lazy checking*: after scrolling through the traceprintf statements from initialization, we see `User Prog brk.c: PID -> 1`, and then a Brk syscall caused by mallocing a ton of memory. Then right below that `Hardware    | Syscall trap Brk`, we see that page frame numbers are being allocated to the region1 heap's virtual address space. Then when we return to `brk.c`, it prints the address returned by malloc. This repeats for 9 times until malloc begins to return NULL. If you patiently scroll far enough to where our Brk fails, you'll see the kernel reporting the failure in Brk:

  ```
  Yalnix     kernel calling Brk(2170880)
  Yalnix     INVALID ADDRESS ::: target -> 137 heap_index -> 124 data_index ->
  2 stack_index -> 127
  ```

- the kernel is reporting an invalid target address index. We then see that our malloc in `brk.c` received NULL, which means that our Brk behaved correctly when it fails also.

Fork - `fork.c`

`progs/fork.c` tests Fork, so we test it by calling `./yalnix progs/fork -W`

`fork.c` simply calls Fork() then makes the parent print something along with its PID, and the child print something along with its PID.

Here's the code to save you from jumping around windows:

```
int main(int argc, char const *argv[]) {
    TracePrintf(1,"fork.c: Calling fork()...\n");
    int rc = Fork();
    int pid = GetPid();
    int ppid = 0;
    if (rc == -1) {
        TracePrintf(1, "\n!!fork.c:Fork Syscall Failed!!\n");
    } else if (rc == 0) {
        TracePrintf(1, "fork.c: hello from child! PID -> %d\tPPID -> %d\n", pid,
ppid);
    } else {
        TracePrintf(1, "fork.c: hello from parent!PID -> %d\tPPID ->
%d\tCHILD_PID -> %d\n", pid, ppid, rc);
    }
    return 0;
}
```

- tracing for this is in `test_traces/FORK_TRACE`make

- *lazy checking*: if you do `cat test_traces/FORK_TRACE | grep fork` you'll see that both the parent and child successfully print the correct pid. (refer to `progs/fork.c` to see the TracePrintf command)
- *not lazy checking*: looking at `test_traces/FORK_TRACE`, the Hardware is detecting the right syscalls, memory allocation seems to be correct when creating the child, just like in lazy checking, the TracePrintfs are coming through and have the right information!

GetPid - `pid_test.c` and every other program

`pid_test` tests GetPid, so we test it by calling `./yalnix progs/pid_test -W`

it's also been done in pretty much all the other test programs.

Exec - `exec1.c` and `exec2.c`

`exec1.c` and `exec2.c` tests Exec, so we test it by calling `./yalnix progs/exec1 -W` or `./yalnix progs/exec2 -W`, the behavior is pretty much the same.

`exec1.c` does not fork, but execs and runs `exec2.c`, meanwhile `exec2.c` execs and runs `exec1.c`, so what we expect to see is some jumping around.

Here's the code to save you from jumping around windows: I'm, showing only `exec1.c` as it differs from `exec2.c` by one line.

```
int main(int argc, char const *argv[]) {
    int pid = GetPid();
    int ppid = 0;
    TracePrintf(1, "exec1.c: PID -> %d\tPPID -> %d\n", pid, ppid);

    // makes log more readable
    Pause();

    Exec("progs/exec2", (char **) argv);
    return 0;
}
```

- tracing for this is in `test_traces/EXEC_TRACE`
- *lazy checking*: running `cat test_traces/EXEC_TRACE | grep exec` will show the print statements from the different files, you can see that the process is successfully alternating between the two programs, you can also see the kernel reporting its syscalls and the arguments that go into it. You can also see LoadProgram reporting the program that its loading.
- *not lazy checking*: `less test_traces/EXEC_TRACE`, and as always, scroll through the traceprintfs that come before pid 1 exits KernelStart. We see a print statement from `exec1.c` then we switch to idle, and then switch back. Once we switch back, we see the kernel report that we're calling exec: `Yalnix     kernel calling Exec(progs/exec2, ...)`. Then right below this, we see that LoadProgram takes in `progs/exec2` and flushes TLB's region1 and rewrites PTBR1. Then we see a lot of deallocating of page frames and allocation of new page frames then some TLB flushes. This is us loading the new program and updating the page tables and flushing the tlb. When we return from the Yalnix handler for TRAP_KERNEL (Exec), we see that the stack pointer, and pc are different, but we have the same PTBR1 and kernel stack frames.

Wait - `wait_exit.c`

`wait_exit.c` and `to_exec.c` test this. `wait_exit.c` forks twice, makes both children call exec on `to_exec.c`, then makes the parent process call `wait` and lets us know when the child process. Within `to_exec.c`, the two child processes will return with different return codes, to see if wait is successfully getting the exit code.

Here's the code for `wait_exit.c`:

```
int main(int argc, char const *argv[]) {
    int pid = GetPid();
    int ppid = 0;

    int fc = Fork();
    if (fc == 0) {
        char *args[2];
        args[1] = "progs/to_exec";
        args[2] = "Child 1";
        Exec("progs/to_exec", args);
    }
    fc = Fork();
    if (fc == 0) {
        char *args[2];
        args[1] = "progs/to_exec";
        args[2] = "Child 2";
        Exec("progs/to_exec", args);
    }

    int rc = 0;
    while (Wait(&rc) != -1) {
        TracePrintf(1,  "wait_exit.c: PID -> %d\tmessage: child has exited with
code (%d)\n", pid, rc);
    }

    TracePrintf(1, "wait_exit.c: Exiting out of wait_exit (pid %d)\n", pid);
    return 0;
}
```

and here's the code for `to_exec.c`

```
int main(int argc, char const *argv[]) {
    int pid = GetPid();
    TracePrintf(1, "to_exec.c: PID -> %d\n", pid);
    if (pid == 2) {
        return 2;
    }
    else {
        return 0;
    }
}
```

- tracing for this is in `test_traces/WAIT_TRACE`
- *lazy testing*: `cat test_traces/WAIT_TRACE | grep wait` shows the traceprintf statements from `wait_exit.c`, which shows that it successfully got the exit code of 2 and 0 from its children. `cat test_traces/WAIT_TRACE | grep to_exec` will show the traceprintf

statements from `to_exec.c`, which shows that the program is being run, that is, exec is working.

- *not lazy testing*: going through the tracefile, the order of the traceprintf statements mentioned above are in the right order, in that both child processes traceprinted from `to_exec` because the parent process finished waiting.