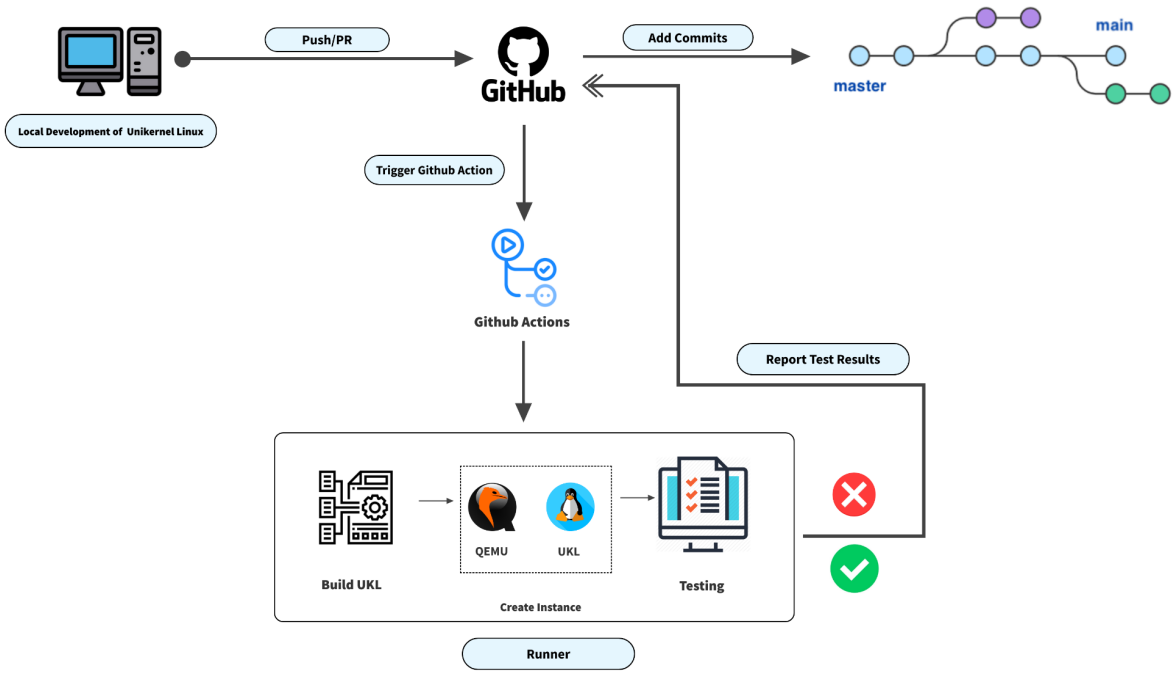
CI for UKL: Project Documentation

I. Project Overview

Our project (*Continuous Integration for Unikernel Linux*) was conducted as part of the EC528/CS528 Course at Boston University from September to December 2021. The goal of this project was to implement a continuous integration (CI) testing framework (using GitHub Actions) for the development of a linux-based unikernel (UKL). Currently, our tests are all designed to run in Github-hosted “Runners" (VMs that contain a runner application that listens for and runs job requests from Github). Github-hosted runners are virtual machines in Azure created and managed by Github, but we include details on how these tests may be modified to run on self-hosted (non-GitHub) machines.



Below, we describe the details of our testing system as well as guidance on how new tests may be added to the framework. We have also included an appendix with details on all changes we made to dependency files (in unikernelLinux/linux, unikernelLinux/ukl, and unikernelLinux/min-initrd) as well as descriptions of the specific testing files within the unikernelLinux/ukl/tests folder.

II. Instructions for Usage

Running Workflows

Currently, both our Unit Tests and Configuration Tests are automatically triggered whenever there is a push or pull request to the ukl-5.14 branch in the unikernelLinux/linux repository. In addition, these tests are scheduled to run daily at 05:00 UTC (~12AM EST). Lastly, one can manually trigger these workflows via a workflow dispatch. To trigger a workflow manually, in the Actions tab of the repository, click on the workflow name and there will be a button next to a banner that says “Run Workflow”.

After a workflow is triggered, the Actions tab displays any past or current workflows that have been run. One can also see the status of a workflow (whether it succeeds or fails), how long ago it was triggered, and how long it took to complete (or ‘In progress’ if it is still running). Additionally, if a workflow run is selected, one can see the jobs that are running within it, and within each job, the different steps (individual tasks run as part of a job). By clicking a step, one can see its output, which is helpful for debugging (especially if the test failed). A failed workflow can easily be re-run by clicking ”re-run all jobs.”

Enabling/Disabling Workflows

Workflows are located in the unikernelLinux/linux repository and can be accessed from the Actions tab in the repository home page. On the left side of the Actions tab, the available workflows are displayed. Disabled workflows are shown with a yellow exclamation symbol, whereas enabled ones have a gray symbol.

- To disable a currently enabled workflow, the name of the workflow has to be selected from the left panel of the Actions tab. Then on the 3 dots (‘…’) button next to the search bar to filter workflow runs, one can access the button to disable a workflow.
- To enable a disabled workflow, select the workflow name from the Actions tab and a yellow banner will appear saying that the workflow has been disabled manually. To enable it, simply click on the “Enable Workflow” button.

III. Testing System Overview

Below, we describe the components of our testing system, including a description of the workflow files used as well as the details of each test we run.

A. Reusable Cloning Script

As each of our tests require checking out various dependency repositories and installing tools required to build the UKL (and boot it in QEMU), we created a single reusable script (unikernelLinux/linux/.github/workflows/composite/clone/action.yml) to perform these tasks. This script is called by each of our workflow scripts.

This script “checks out” (clones) each of the dependency repositories, creates the directory structure** required for building/running the UKL, and installs the various Ubuntu tools needed to compile the UKL and run it in QEMU. As several of the dependency repositories are currently private, a token (provided by the workflow which calls this script) is required to “checkout” (clone) these repositories. Should different repositories or branches be required in the future, a user can simply update the values for “repository” or “ref” respectively.

**Note: The directory structure created by this script is a special design for the UKL, which is somewhat “non-standard” in GitHub Actions. As such, we provide additional explanation below:

- In order for the latest changes (from a pull request) to be run in a workflow, the first repository checked out must be the repository for which the workflow is created (i.e., the repository in which the workflow files are stored). This repository becomes the top-level directory in the virtual machine in which the workflow is run (under the hood, GitHub Actions sets this directory as the “GITHUB_WORKSPACE” environment variable). In our case, to get the latest changes from the Linux repository, we need to first checkout the ukl-5.14 branch of unikernelLinux/linux.
- However, for building the UKL, we actually need to make the linux repository a subdirectory of the ukl repository.
- As such, the second step in this Workflow (“Copy linux dir into ukl dir”) moves the linux directory into a subdirectory of the ukl directory. This step then allows us to use the normal makefiles for compiling/running the UKL.
- Because we make this change to the directory structure (which is not a normal procedure in GitHub Actions), we then have to add a final step at the end of every of job in our workflow files (labelled as “Prepare for Cleanup”) which moves the linux directory back to the parent directory of “ukl”. This step is required since, after a job completes, the GitHub Actions runner runs a few post-job clean-up procedures, one of which looks for this specific script (which is inside the linux directory). If we do not move linux back to the parent directory (where GitHub will look), the clean-up procedures will not be able to find this script and the job will fail.

B. UKL Unit Tests Workflow

Our UKL_UNIT_TEST workflow runs several separate unit tests. Each test is run on both pushes or pull requests to the ukl-5.14 branch in this Linux repository (as well as nightly via a scheduled workflow run). Moreover, each test is run as a separate “job”, meaning that each test is run in parallel on its own GitHub-hosted virtual machine.

Within this workflow, the first 4 steps -- as well as the last step -- of each “job” are the same (with the exception of the memcached test):

- The first step checks out the “ukl 5-14” branch of the unikernelLinux/linux repository.
- The second step calls the reusable cloning script to clone all dependency repositories, set up the directory structure, and install Ubuntu tools for compiling the UKL/running QEMU.

- The third step replaces the standard linux .config file with a copy of the configuration file (located in ukl/tests) in which “UKL_CREATE_AFTERSPACE” is enabled.
- The fourth step simply runs “make all” to compile the UKL
- As mentioned in the description of the Reusable Cloning Script above, the last step of each job simply performs directory restructuring to enable the runner’s default cleanup processes to complete successfully.

Below, we describe the tests run in this workflow:

Boot Test

- The Boot Test simply tests whether the compiled UKL is able to be booted in QEMU.
- This test is run by using a shell script (run_boot.sh), which compiles the test application (a simple c file that prints a magic string and exits) into a unit_test.o object, builds the UKL, and boots QEMU (with QEMU’s serial output fed into a file called “test.out”).
- After the test is run, the shell script checks the output from QEMU to ensure that the magic string was printed.
    - If the check passes, the script prints a success message and exits.
    - If the check fails, the script prints the QEMU output file to console (for usage in debugging) and performs an “exit 1” (which will cause the job to fail).

LEBench_SysCalls_Tests

- The “LeBench System Calls Test” checks whether the various system calls run in lebench (read, write, epoll, etc.) are able to run successfully.
- The test is run by using a shell script (run_lebench_syscalls.sh) which compiles the test application (a modified version of lebench that prints a different magic string after each system call test is run) into a unit_test.o object, builds the UKL, and boots QEMU (with QEMU’s serial output fed into a file called “test.out”).
- After the test is run, the shell script checks the output from QEMU to ensure that each magic string was printed.
    - If the check passes, the script prints the results and exits.
    - If the check fails, the script prints the QEMU output file to console (for usage in debugging), prints the results (which tests passed and which tests failed), and performs an “exit 1” (which will cause the job to fail).

Multi_Thread_Tests

- The Multi-Thread test ensures that a multithreaded program (in this case, a C program using the POSIX thread (pthread library)) can successfully run in the UKL.
- The test is run by using a shell script (run_pthread.sh) which compiles the test application into a unit_test.o object, builds the UKL, and boots QEMU (with QEMU’s serial output fed into a file called “test.out”).
- In this case, the test application simply creates 100 threads (each of which does a simple task), joins them, and then prints a magic string if the program completes successfully.

- After the test is run, the shell script checks the output from QEMU to ensure that the magic string was printed.
    - If the check passes, the script prints the results and exits.
    - If the check fails, the script prints the QEMU output file to console (for usage in debugging), prints the results, and performs an “exit 1” (which will cause the job to fail).

Memcached_Tests

- The memcached tests check whether we can write and read to a memcached server running in the UKL (which allows us to test UKL networking capabilities. For this test, we use both memaslap and telnet to verify that we can write and read to the memcached server in the UKL. Unlike the other unit tests described above, this job requires some additional setup:
    - First, the script installs libmemcached from a tarball saved in ukl/tests/memcached, in order to provide a tool to use the memaslap command (note: before installation, the script updates 2 of the source files in the tarball to ensure compatibility with the latest gcc compiler).
    - It then checks out both the unikernelLinux/libevent (ukl branch) and unikernelLinux/memcached (ukl branch) repositories and compiles them. It then compiles/builds the UKL.
- Following these set-up steps, the job runs a shell script (run_memcached_server.sh) that starts the memcached server as a background process (and redirects the output to a file called “qemu_memc_out”). It also sleeps for 30 seconds, as we found we needed to wait some time for QEMU to finish booting before attempting to write and read to memcached.
- Then, to run the memaslap tests (bulk writes and reads to memcached), the job executes a shell script (run_memaslap_tests) which first runs the memaslap test for 30 seconds with only 1 thread and then it runs a second test with 16 threads for another 30 seconds. The script then checks the output from these two memaslap tests (stored in a file) and verifies that more than 0 bytes were able to be read/written.
    - If the check passes, the script prints the results and exits.
    - If the check fails, the script prints the QEMU output file to console (for usage in debugging), prints the results, and performs an “exit 1” (which will cause the job to fail).
- Then, the test runs a second script (run_telnet_tests) to use telnet to connect to the memcached server and execute several set/get commands (provided in telnet_commands) to test storing and retrieving specific key/value pairs. The output from this test is stored in a file, which is subsequently checked to ensure that the values for the keys retrieved via “get” were the same as those that were “set.”
    - If the check passes, the script prints the results and exits.
    - If the check fails, the script prints the QEMU output file to console (for usage in debugging), prints the results, and performs an “exit 1” (which will cause the job to fail).

- After the tests complete, the script kills QEMU by extracting its PID and executing “kill” on the PID (since unlike the other unit tests, memcached will continue running forever within QEMU).

C. UKL Configurations Tests Workflow

Our UKL_CONFIGS_TEST workflow is used to ensure that the various combinations of the following three UKL-specific configurations are able to compile and run:

- UKL_USE_RET
- UKL_SAME_STACK
- UKL_USE_IST_PF

Note: For these tests, we **always** enable the “UKL_CREATE_AFTERSPACE” configuration.

Like the Unit tests, this workflow is triggered on a push or pull request to the ukl-5.14 branch (as well as nightly via a scheduled workflow run), and each test is run as a separate job on its own GitHub-hosted virtual machine. All jobs within this workflow follow the same steps and the only difference between them is the combination of the UKL specific configurations that are enabled in each (which are specified in the third step below). The steps for each job in this workflow are as follows:

- The first step checks out the “ukl-5.14” branch of the unikernelLinux/linux repository.
- The second step calls the reusable cloning script to clone all dependency repositories, set up the directory structure, and install Ubuntu tools for compiling the UKL/running QEMU.
- The third step replaces the standard linux .config file with a copy of the configuration file (located in ukl/tests), and enables a specific combination of the UKL configurations.
- The fourth step simply runs “make all” to compile UKL
- The fifth step tests running LEBench_SysCalls_Tests with the specified configuration options enabled.
- As mentioned in the description of the “Reusable Cloning Script” above, the last step of each job simply performs directory restructuring to enable the runner’s default cleanup processes to complete successfully.

The jobs run in this workflow are detailed below:

TEST_NO_UKL_OPTIONS_ENABLED:

- As the name suggests, this job tests running the LEBench_SysCalls_Tests with no specific UKL configuration options (other than UKL_CREATE_AFTERSPACE) enabled.

TEST_ALL_UKL_OPTIONS_ENABLED:

- This job tests running the LEBench_SysCalls_Tests with all UKL configuration options enabled.

TEST_UKL_USE_SAME_STACK_ONLY

- This job tests running the LEBench_SysCalls_Tests with only the UKL_USE_SAME_STACK and UKL_CREATE_AFTERSPACE options enabled.

TEST_UKL_USE_RET_ONLY

- This job tests running the LEBench_SysCalls_Tests with only the UKL_USE_RET and UKL_CREATE_AFTERSPACE options enabled.

TEST_UKL_USE_IST_ONLY

- This job tests running the LEBench_SysCalls_Tests with only the UKL_USE_IST_PF and UKL_CREATE_AFTERSPACE options enabled.

TEST_UKL_USE_SAME_STACK_and_UKL_USE_RET

- This job tests running the LEBench_SysCalls_Tests with the UKL_USE_SAME_STACK, UKL_USE_RET, and UKL_CREATE_AFTERSPACE options enabled.

TEST_UKL_SAME_STACK_and_UKL_USE_IST

- This job tests running the LEBench_SysCalls_Tests with the UKL_USE_SAME_STACK, UKL_USE_IST_PF, and UKL_CREATE_AFTERSPACE options enabled.

TEST_UKL_USE_IST_and_UKL_USE_RET

- This job tests running the LEBench_SysCalls_Tests with the UKL_USE_RET, UKL_USE_IST_PF, and UKL_CREATE_AFTERSPACE options enabled.

D. Latency Tests Workflow:

The LATENCY_TEST is designed to allow a user to automatically run tests to capture and display the latency of specific operations within the UKL.

As a proof of concept (POC), we have implemented a single test designed to capture the latencies of the various system call tests run in lebench. A description of this test is below:

- The “LeBench Latency Test” simply captures and displays the time required for the various system call tests run in lebench (read, write, epoll, etc.) to complete.
- The test is run by using a shell script (run_lebench_latency.sh) which compiles the test application (a modified version of lebench that records the elapsed time of each test) into a unit_test.o object, builds the UKL, and boots QEMU (with QEMU’s serial output fed into a file called “test.out”).
- After the test is run, the shell script executes a python file (logsReader.py), which parses the output from QEMU (stored in a file called “test.out”) to extract the total time for each test.
    - If all tests have run (the Python file is able to extract total time for each test), the latencies for each test are printed to the console and the test exits successfully.

    - If not all tests have run (the Python file is not able to find results for all tests), the script prints the QEMU output file to console (for usage in debugging), prints the results (which tests passed and which tests failed), and performs an “exit 1” (which will cause the job to fail).

To experiment with how the latencies captured by this test might be influenced by the machine on which the test is run, we ran the test in 3 separate environments:

- GitHub-provided VM (in Azure): We ran this program a few times in Azure and noted that the entire test program tended to take ~10 minutes to complete with no individual system-call test taking longer than 115 seconds (in all cases, the ‘Write Test” took the longest).
- Self-hosted VM in MOC: In our MOC VM, the test program took ~20 minutes to complete (based on our observations from 3 separate runs) with some individual system call tests taking in excess of 3 minutes to complete.
- Physical Server in CloudLab: In Cloudlab, we ran the test 3 separate times, and the test program took less than 8 minutes to complete in each run (with no individual system call test lasting longer than 95 seconds). Interestingly, the compilation of the UKL (which takes approximately 40 minutes in Azure and ~1 hour in our MOC VM) took less than 10 minutes on the physical server.

Currently, this test is available in the UKL’s repository Actions tab but is not enabled as the project does not have a stable physical server to allocate for running these tests. Should the project wish to activate this test in the future:

- The workflow will have to be enabled as explained in the Enabling/Disabling workflows section.
- Once the workflow is enabled, it will be available to trigger manually (via the workflow dispatch option) by pressing the run workflow button. To trigger the test via pushes or pull requests, and/or schedule it to run daily, the event activities lines (lines 11-14 and 17-18) in the LATENCY_TEST YAML file should be uncommented.
- Lastly, the test is currently configured to run on a GitHub-Hosted machine (in Azure). To instead run this test on a self-hosted machine, the “runs-on” line (line 31) should be changed to “self-hosted” and the self-hosted machine should be configured as explained on the Running tests on self-hosted machines section.

III. Guidance for Adding New Tests

Below, we provide some guidance on how new tests may be added to this testing system. These instructions assume some familiarity with GitHub Actions. For more detailed documentation on using GitHub Actions, please refer to the GitHub actions tutorial document.

Adding New (Simple) Unit Tests

Adding additional (simple) unit tests that do not require specialized settings for QEMU (e.g., special networking or command line options) can be accomplished as follows:

- Add the source code for the application as a subfolder within the tests directory of the actions-files branch of unikernelLinux/ukl.
- Add a new target for this test to the Makefile within the tests directory that compiles the application into an object file called unit_test.o.
- Create a shell script (we would suggest using run_boot.sh or run_lebench_syscalls.sh as a template) to automate compilation, running the test in QEMU, and checking test output.
- Update the UKL_UNIT_TEST file with a new job to run this test (the easiest method is to simply copy an existing job and change the name of the shell script to be run).

Creating More Complicated Unit Tests

To create more complicated tests (i.e., those which might require different arguments provided to QEMU), we recommend looking at the Makefile and workflow job for the memcached test (in UKL_UNIT_TEST).

From our experience, the easiest method is to similarly store all source code and shell scripts in the tests directory of the actions-files branch of unikernelLinux/ukl, but then add new (test-specific) targets for this test in the Makefile in the ukl directory as well as the Makefile in the min-initrd directory which are both cloned by our reusable cloning script used by the workflows.

From experience, we recommend first getting these tests to run on an Ubuntu virtual machine before porting any source code/shell scripts to GitHub. The primary reason is that a GitHub workflow will require approximately 35-40 minutes to build the kernel each time that it is run, while local testing can simply use a pre-built kernel (and use make linux-build to recompile the UKL with the test application).

Running Tests in Self-Hosted Runners

To set up self-hosted runners, we first recommend reviewing the latest documentation on using self-hosted runners in GitHub, which is available here.

After securing a machine on which tests can be run, we recommend below steps for setting up this machine as a self-hosted runner for GitHub Actions:

- Run:
    - sudo apt install liblttng-ust0 libkrb5-3 zlib1g libssl1.1
- On Github.com go to Settings -> Actions -> runners -> new self-hosted runner runner (make sure to select linux, x64 for Ubuntu)
    - Copy and paste all the commands into the instance

    - Make sure the last command begins listening and connects to GitHub successfully. If this step is successful, you can then use ctrl^c to stop listening.
- Install as a service:
    - Run:
        - sudo ./svc.sh install
    - Start service by running:
        - sudo ./svc.sh start

Then, in the workflow script of whichever job you wish to run on your self-hosted runner, change the value for runs-on to self-hosted (e.g., change from *runs-on: ubuntu-latest* to *runs-on: self-hosted*).

Appendix

A. Summary of Changes to UKL Dependency Directories

To enable our tests to run in GitHub Actions (and specifically on Ubuntu, which is the Linux distribution used in Github-hosted runners), we needed to modify several of the existing files required to compile and build the UKL. A summary of these changes are included below:

unikernelLinux/Linux

While running our initial tests in GitHub Actions, we ran into an issue where (with the UKL_AFTERSPACE configuration enabled) after the application would run and the UKL would exit, QEMU would remain running. To solve this issue, we set the “no-reboot” flag when starting QEMU and created a “shutdown” script in our root file system (see “shutdown” in the unikernelLinux/min-initrd section below) which simply forces QEMU to shutdown (since the “-no-reboot” flag is set).

To run this script, we edited line 1516 of unikernelLinux/Linux/init/main.c to call the shutdown script (from the root file system) when “after space” is triggered (previously, the “init” script was simply called again).

An alternative to this change is to simply use the timeout command when running tests in QEMU (e.g., “sudo timeout 10m make run”) so that QEMU is terminated after running for a certain amount of time. As we already do make use of timeouts in our test scripts, the tests will run if the change described above is reverted.

unikernelLinux/ukl

We made the following updates to files in unikernelLinux/ukl (updates are in actions-files branch):

- Makefile:
  - To compile on Ubuntu, we added the “-fno-pic” flag to both the `LEBench_UKL_FLAGS` and `UKL_FLAGS`
  - We also added “-fno-pic”, “no-pie”, “-nostartfiles” flags to all calls to “make” in the “gcc-build” target in order to compile libgcc on Ubuntu
  - The “-fno-pic” was also added to the “undefined_sys_hack.o” target for Ubuntu compilation
  - The following targets were added to the Makefile in order to facilitate building via GitHub Actions:
    - **unit_test**
      - This target compiles a single object file (called “unit_test.o”) into a “unit_test.ukl” file which is then used to create the UKL.a (unikernel file)

            - This target is used by our boot test, lebench sys calls test, lebench latency test, and multi-threading test to compile each of these applications into an object file called “unit_test.o”. We implemented this target so that any file called “unit_test.o” could be used to build the UKL without requiring a custom target in the Makefile
        - **libevent**
            - This target is used to build libevent (which is required by memcached for our memcached test).
            - The target is based on a previous Makefile used for compiling memcached for the UKL, but for Ubuntu compilation, we added the “-fno-pic” and “-no-pie” flags to the ”CFLAGS” used for libevent configuration
        - **memcached**
            - This target is used to build memcached (for our memcached test) and again is based on a previous Makefile used for compiling memcached for the UKL.
            - Again, we added the “-fno-pic” and “-no-pie” flags to the ”CFLAGS” used for memcached configuration
            - Moreover, due to pathing issues, we appended “memcached/” (the directory name) to the beginning of each of the object files during linking (which is called from the ukl directory -- the parent directory of memcached)
        - **run_memcached**
            - This target is used to call the runU_memcached target in the minitrd makefile (see description below) which boots QEMU with memcached-specific command line options
    - Finally, all cloning operations in this Makefile were removed and changed into “Checkout Actions” in the Reusable Cloning script due to some permissions issues (since some of the dependency repositories are private) that we faced when cloning from the Makefile in a workflow
- clean_build.sh
    - For Ubuntu compilation, we added the “-fno-pic” flag to the CFLAGS provided to glibc/configure
- tests/
    - This folder was added to encapsulate the various unit tests run in our workflows. The files in this folder are detailed in the Description of Test Source Files section below.

unikernelLinux/min-initrd

The following files were updated in unikernelLinux/min-initrd (updates are in the actions-files branch):

- Makefile:
    - “mount” was added to the Supermin PACKAGES to allow files to be mounted during the “init” process
    - We manually set the shell (SHELL=/bin/bash) in order to ensure that bash was used in executing the makefile (/bin/sh defaults to dash in Ubuntu which cannot parse the “if [...]” syntax in the Makefile)
    - We set SMP (number of cores) to 1 given that QEMU must run without KVM in GitHub Actions (and using multiple cores caused the CPU to occasionally lock up when using multiple threads)
    - We updated the “QEMU” and “options” environment variables to run without KVM (as KVM cannot be used in GitHub Actions)
    - We changed the -m (memory) flag to 3g (which we found was the maximum amount of virtual memory we could create in QEMU when running on our Ubuntu virtual machines)
    - We also added the “-no-reboot” flag to the QEMU options in order to force QEMU to exit when reboot is called from our shutdown script.
    - We updated the DISPLAY arguments with “-serial file:”../test.out” so that all output from QEMU would be piped to a file called “test.out” (in the ukl directory) when running tests in GitHub Actions. This allows us to parse the file and search for magic strings printed by our unit tests
    - For both the COMMANDLINE and COMMANDLINE_MEMC options, we added “nosmep” to fix a bug where the user space code executing in kernel mode (which is the expected behavior of the UKL) was causing QEMU to crash
    - We added the following variables and targets for running our memcached tests:
        - DISPLAY_MEMC -- The display options for the memached test. The serial output of QEMU is fed to standard output (although our test then pipes this output into a file)
        - COMMANDLINE_MEMC -- The command line options for the memcached test. The key additions here were a user (ukl_user) and port (11255) to be used by memcached
        - NETWORK_MEMC -- These options create a very simple network for the memcached test where tcp traffic is forwarded to port 11255
        - runU_memcached - This target runs QEMU with the special memcached options described above
    - In addition, we added a target for supermin.d/shutdown.tar.gz
        - This simply zips our shutdown script (described below) and adds it to the root file system in the $(TARGET)/root target
- init:
    - The principal change we made to the init file was the insertion of three lines to create a dummy user (ukl_user) in the “etc” folder so that this user could be used to run memcached
- shutdown:
    - The shutdown script simply forces QEMU to reboot. However, since the no-reboot flag is set in the options we provided when booting QEMU, QEMU will

actually shut down. The primary purpose of this script is to stop QEMU after a test program runs in the UKL (otherwise, QEMU will remain running after the UKL exits). An alternative to using this script is simply using a timeout when starting QEMU (10 minutes is sufficient for most tests other than the latency tests, which requires at least 20 minutes in some cases).

B. Description of Test Source Files

The various shell scripts, Makefiles, and application source code utilized by the workflows within our testing system are stored in the tests folder within the unikernelLinux/ukl repository (in the actions-files branch). Below, we describe each of the files and folders within this tests folder.

Files

- Makefile
    - The Makefile within the test folder contains targets for the boot_test, lebench_syscalls_test, lebench_latency_test, and pthread_test.
    - This makefile is used to compile each of these tests into a “unit_test.o” file, which can then be used by the Makefile within ukl to compile the UKL with this test application.
- linux_config
    - This linux configuration file is a direct copy of the golden_config-5.7-broadcom used for normal UKL compilation, but with the “UKL_CREATE_AFTERSPACE” configuration set.
    - This file is used to replace the normal .config file when building Linux in our workflows so that UKL_CREATE_AFTERSPACE is always enabled.
    - It is also used as the base file to which different combinations of UKL configuration options are added on the UKL Configurations Tests.
- run_boot.sh
    - This shell script is used to run the boot test.
    - The script compiles the boot test application (a simple c file that prints a magic string and exits) into a unit_test.o object, builds the UKL, and boots QEMU (with QEMU’s serial output fed into a file called “test.out”)
    - After the test is run, the shell script checks the output from QEMU to ensure that the magic string was printed.
        - If the check passes, the script prints a success message and exits.
        - If the check fails, the script prints the QEMU output to console (for usage in debugging) and performs an “exit 1” (which will cause the job to fail).
- run_lebench_syscalls.sh
    - This script is used to run the lebench system calls test
    - The script compiles the lebench_sycalls_test application (a modified version of lebench that prints a different magic string after each system call test is run) into a unit_test.o object, builds the UKL, and boots QEMU (with QEMU’s serial output fed into a file called “test.out”).
    - After the test is run, the shell script checks the output from QEMU to ensure that each magic string was printed.
        - If the check passes, the script prints the results and exits.
        - If the check fails, the script prints the QEMU output file to console (for usage in debugging), prints the results (which tests passed and which tests failed), and performs an “exit 1” (which will cause the job to fail).
- run_pthread.sh

    - This script is used to run the Multi_Thread_Tests.
    - The script compiles the pthreads test application into a unit_test.o object, builds the UKL, and boots QEMU (with QEMU's serial output fed into a file called "test.out").
    - In this case, the test application simply creates 100 threads (which each perform a simple task), joins them, and then prints a magic string if the program runs successfully.
    - After the test is run, the shell script checks the output from QEMU to ensure that the magic string was printed.
        - If the check passes, the script prints the results and exits.
        - If the check fails, the script prints the QEMU output file to console (for usage in debugging), prints the results, and performs an "exit 1" (which will cause the job to fail).
- run_lebench_latency.sh
    - This script is used to run the lebench latency test.
    - The script compiles the test application (a modified version of lebench that records the elapsed time of each test run within the program) into a unit_test.o object, builds the UKL, and boots QEMU (with QEMU's serial output fed into a file called "test.out").
    - After the test is run, the shell script executes a python file (logsReader.py), which parses the output from QEMU to extract the total time for each test.
        - If all tests have run (the Python file is able to extract total time for each test), the latencies for each test are printed to the console and the test exits successfully.
        - If not all tests have run (the Python file is not able to find results for all tests), the script prints the QEMU output file to console (for usage in debugging), prints the results (which tests passed and which tests failed), and performs an "exit 1" (which will cause the job to fail).
- logsReader.py
    - This is the python script used by the lebench latency test, which reads the QEMU output file and extracts the running time for each test.
    - If the script is able to extract running times for each test, it prints the results and exits normally.
    - If the script is not able to extract running times for some tests, it prints the results (including which tests did not run) and exits with error code 1 in order to trigger a test failure.
- run_memcached_server.sh
    - This shell script is used by the memached (networking) tests.
    - It starts the memcached server as a background process (and redirects the output to a file called "qemu_memc_out").
    - It then sleeps for 30 seconds, to allow QEMU sufficient time to boot (before any additional tests are run)

Folders

- /boot:
    - The boot folder stores the source file (boot_test.c) that is used by the Boot Test to check that QEMU can successfully boot the UKL.
    - It simply prints a magic string ("UKL Booted Successfully") and exits.
- /lebench:
    - The lebench folder contains two files:
        - **lebench_latency_test.c**: This file is used as the application source code for the lebench latency test. It contains multiple functions that make different system calls. This file was provided to our team (as mybench_small) by our mentors, and we modified it by adding print statements to print the latencies of each test.
        - **lebench_syscalls_test.c:** This file is used as the application source code for the lebench system calls tests. It contains multiple "test" functions that make different system calls (again, this is a slightly modified version of the mybench_small file provided by our mentors) . After each system call test runs, we print a magic string which we use to ensure that the system call was able to complete successfully.
- /memcached:
    - The memcached folder contains the different files we used in order to automate the compilation of memcached and subsequent testing using memaslap and telnet. These files are all used by the memached test.
        - **libmemcached-1.0.18.tar.gz:** This tar file contains the source code for libmemcached (which contains the memaslap tool). We untar the file (and compile/install it) to use memaslap in the memcached test.
        - **Makefile.in:** In order to use memaslap, we had to modify the makefile used by libmemcached (by adding additional flags to enable compilation on newer versions of gcc). This file is the modified version of the original libmemcached makefile that we copy into the libmemcached-1.0.18 directory (to replace the original Makefile) when installing memaslap as part of the memcached test.
        - **memflush.cc:** memflush.cc is another file from libmemcached-1.0.18 that we had to modify in order to compile memaslap with newer versions of gcc. We use this file to replace the original memflush.cc after untarring the Libmemcached-1.0.18.tar.gz file.
        - **run_memaslap_tests.sh**: This shell script file runs the memaslap tests.
            - It first runs the memaslap command (which executes multiple reads and writes) with 1 thread for a period of 30 seconds and stores the results in an output file. It then checks the output file to see if the read bytes and written bytes that the command returns are not 0.
            - It then repeats the same test using 16 threads for a period of 30 seconds.

        - If both tests are able to read/write more than 0 bytes, the script prints a success message and exits normally.
        - If either of the tests are not able read/write more than 0 bytes, the script prints the entire QEMU output, results from the output file, and test results to console (for debugging) and exits with error code 1, which triggers the job to fail.
    - **run_telnet_tests.sh**: This shell script runs secondary tests for memcached by using the telnet tool.
        - The script uses telnet to connect to the memcached server rin the UKL (running in QEMU) and then runs series of set/get commands that are stored in the script telnet_commands.sh (and outputs the results of the commands to an output file called “telnet_tests.out”).
        - After executing these commands, it reads the output and checks whether the key/value pairs it retrieved were the ones it had set.
        - If all retrieved key/value pairs match those that were stored, the script prints a success message and exits normally.
        - If any key/value pairs do not match, the script prints the entire QEMU output and test results to console (for debugging) and exits with error code 1, which triggers the job to fail.
    - **telnet_commands.sh:**
        - This shell script is invoked by the script run_telnet_tests.sh and is used to simply execute a number of “set” and “get” commands to respectively store and retrieve key/value pairs from memcached using telnet. It also tests updating the value of a key in memcached and retrieving it to verify that it was updated correctly.
- /pthreads:
    - The pthreads folder stores a single source file (pthread_test.c) that is used by the multithreading test to check that the UKL can run a multithreaded application.
    - This test application simply creates 100 threads (each of which do a simple task), joins them, and then prints a magic string if the program runs successfully.