

CS 452 Kernel 3

Benjamin Zhao, Kyle Verhoog

Feb 05, 2018

Contents

Operation	2
Structure	2
Implementation	2
Stack & Memory Layout	3
User Stacks	3
Task Descriptor	3
Task Identification	3
TidTracker	4
Context Switch	4
Kernel to User	4
User to Kernel (on IRQ or SWI)	5
Limitations	6
Scheduling	7
Messaging	7
Send	7
Receive	7
Reply	8
Note on Error Handling	8
Name Server	8
RegisterNS	8
GetNS	8
Implementation	8
RegisterAs	8
WhoIs	9
IRQ Context Switch	9
Kernel To User Task	9
User Task to Kernel	9
Interrupt Handling	9
Interrupt Handler	9
Interrupt Matrix	10
Clock Server	10

Event Types	10
Implementation	10
Clock Server Notifier	11
Syscalls	11
Output	12
Raw Output (Snippet)	12
Explanation	13
User Tasks Output	13
Metrics Output	14
IdleTask	14
IdleTaskInterface	15
Source Code and Hashes	16

Operation

The ELF file is located at `/u/cs452/tftp/ARM/ktverhoo/kernel3.elf` and can be initiated using the typical `load` command.

The kernel should run the `FirstUserTask` and then exit printing usage statistics.

Structure

Implementation

Our implementation follows the same basic loop shown in class:

```
int main(void) {
    initialize();
    while(true) {
        TaskDescriptor *td = schedule();

        if (!td) break;

        TaskRequest req = activate(td);
        handle(td, req);
    }
}
```

Stack & Memory Layout

We implement our stacks growing downward in memory. The kernel stack begins in the middle of the 32MB of memory and the user stacks start at the top of memory.

User Stacks

The user stacks start at the top of memory and are each (for now) 1MB in size. So each consequent user stack is 1MB separated in memory and therefore, there can only be 16 active tasks. Of course, this number is arbitrary for now, and will change once we determine the number of tasks needed for our kernel.

The memory layout is defined in `/include/kernel/kernel.h`. Here we define where the kernel and user stacks begin and how large user stacks are.

Task Descriptor

The task descriptor contains the following fields:

- **tid**: explained in the next section
- **sp**: user task stack pointer
- **psr**: user task status register
- **task**: pointer to function of task
- **parent**: pointer to parent task descriptor
- **next**: pointer to next task descriptor in priority queue
- **priority**: the tasks priority
- **ret**: the value to be returned to the task on syscall
- **sendq**: A circular buffer of size equal to the number of tasks

See: `include/task/task.h` for details.

Task Identification

Tasks are uniquely identified by tids which are tracked as unsigned 32-bit integers. Every task has a unique tid, and currently all zombie tasks return the tid upon exiting. The upper 16 bits of the tid represent a version number and the lower 16 bits represent an id. All tids start at version 0.

For example, the tid `0x0000 0001` represents a task with id 1 and version 0.

See: `include/task/task.h` for details.

TidTracker

The TidTracker is a distributor which distributes unique tids upon request. The tids are pre-generated when the kernel starts running. Tids are re-used (with an incremented version) when a task exits and the tid is returned to the distributor for re-use. When the distributor runs out of tids, likely either two things have happened:

- All tids are in use
- The use of a tid has exceeded 2^{16} re-issues in which an overflow may cause undefined behaviours

The TidTracker uses a circular buffer, prefilled with some maximum number of tids allowed to be allocated to tasks at once. When a task is created, the kernel requests a tid using `tt_get()` from the TidTracker. The tracker then takes the first tid from the queue, pops it and gives it to the kernel. When the task exits, the kernel calls `tt_return()` to return the tid to the tracker. The tracker appends 1 << 16 to the tid, and inserts to the end of the buffer.

```
int tid = tt_get(&tid_tracker);
tt_return(td->tid, &tid_tracker);
```

See: `include/task/task.h` for details.

The circular buffer is implemented using a fixed-sized array, with a start and end index pointing to the head and tail of the queue respectively. The queue has constant $O(1)$ time insertion as well as deletion of head. The circular buffer itself does not have any overflow guards, however we rely on the limited number of tids to ensure we never reach an overflow.

See `src/lib/circularBuffer/circularbuffer.c` for details.

Context Switch

Instead of rephrasing the context switch, here is an annotated version of the function `activate` which handles both the kernel to user and user to kernel switches.

The `activate` function runs a set of inline assembly macros which perform the saving of states to stacks, register manipulation, priviledge changes and jumps to and from user land.

Kernel to User

```
if(td->it) {
    PUSH_STACK("r0-r12, lr");
    WRITE_SPSR(td->psr);
```

```

SET_CPSR(SYSTEM_MODE);
WRITE_SP(td->sp);
POP_STACK("lr");
asm("mov r8, sp");
asm("add sp, sp, #56");

SET_CPSR(KERNEL_MODE);
asm("ldmfd r8, {r0-r12, lr}");
asm("SUBS pc, lr, #4");
}else{
PUSH_STACK("r0-r12, lr");
asm("mov r8, %0"::"r"(td->ret));
PUSH_STACK("r8");
WRITE_SPSR(td->psr);

SET_CPSR(SYSTEM_MODE);
WRITE_SP(td->sp);
POP_STACK("lr");
asm("mov r8, sp");
asm("add sp, sp, #56");

SET_CPSR(KERNEL_MODE);
asm("ldmfd r8, {r0-r12, lr}");
POP_STACK("r0");
REVERSE_SWI();
}

```

User to Kernel (on IRQ or SWI)

```

asm("IRQ_ENTRY:");
asm("stmfd sp, {r0-r12}");

asm("mov r9, sp;"
    "mov r10, lr;");

SET_CPSR(SYSTEM_MODE);

asm("ldmdb r9!, {r0-r7}");
asm("stmdb sp!, {r0-r7, r10}");
asm("ldmdb r9!, {r0-r4}");
asm("stmdb sp!, {r0-r4}");
PUSH_STACK("lr");

SET_CPSR(KERNEL_MODE);

```

```

POP_STACK("r0-r12");
SET_CPSR(SYSTEM_MODE);
READ_SP(td->sp);
SET_CPSR(IRQ_MODE);
READ_SPSR(td->psr);
SET_CPSR(KERNEL_MODE);
td->it = 1;

asm("mov r0, #12"); //ENUM - TR_IRQ
POP_STACK("lr");
asm("b ACTIVATE_END");

asm("KERNEL_ENTRY:");
asm("stmfd sp, {r0-r12}");

asm("mov r9, sp;"
    "mov r10, lr;");

SET_CPSR(SYSTEM_MODE);

asm("ldmdb r9!, {r0-r7}");
asm("stmdb sp!, {r0-r7, r10}");
asm("ldmdb r9!, {r0-r4}");
asm("stmdb sp!, {r0-r4}");
PUSH_STACK("lr");

SET_CPSR(KERNEL_MODE);
POP_STACK("r0-r12");
SET_CPSR(SYSTEM_MODE);
READ_SP(td->sp);
SET_CPSR(KERNEL_MODE);
READ_SPSR(td->psr);
td->it = 0;
SWI_ARG_FETCH("r0");
POP_STACK("lr");

asm("ACTIVATE_END:");

```

See include/asm/asm.h and src/kernel/kernel.c for details.

With our implementation of the context switch all three of the link registers are saved and restored correctly.

Limitations

Number of Mode Switches

As depicted above you can see that there are a number of changes in mode. This could have potential performance issues and is probably an indicator that we should refactor.

Switching Modes

Currently `SET_CPSR(MODE)` is dependent on the usage of a register, namely `r12`. This means that when we re-enter the kernel, we must switch to system mode to access the user stack pointer, corrupting `r12`.

Scheduling

Scheduling is done by managing a set of task queues. There are 32 priorities and hence 32 task queues. Tasks are placed in a task queue corresponding to its priority. The next task that is scheduled is the one at the head of the highest non-empty priority queue.

A 32-bit integer is used to maintain state information about which priority has tasks available. When the i -th bit is flipped, then there are tasks available in the priority i queue.

Refer to `k1.pdf` for more information.

Messaging

Messaging is done using `Send`, `Receive` and `Reply`. The implementations of these is similar to as described in class and the notes.

Send

```
int Send(int tid, void *msg, int msg_len, void *reply, int reply_len);
```

Sends a message to a receiver task `tid` by copying `msg` to the receiver's `msg`. If there is no receiver waiting, the sender is placed in the receiver's `sendq` and blocked.

Receive

```
int Receive(int *tid, void *msg, int msg_len);
```

Receives a message from a sender `*tid` into `msg`, or waits until there is one.

Reply

```
int Reply(int tid, void *reply, int reply_len);
```

Meant to be called from a receiver to return results back to a sender. `reply` is copied from the receiver to the sender.

Note on Error Handling

Due to time limitations there is little to no error handling done around messaging, besides asserts. It is in our backlog to implement error handling for messaging.

Name Server

The name server is implemented as a user task with a couple special system calls which allow it to initialize with the kernel.

RegisterNS

```
int RegisterNS();
```

Registers the calling task with the kernel as the nameserver.

GetNS

```
int GetNS();
```

Returns the tid of the current nameserver registered with the kernel.

Implementation

The Nameserver uses a basic `<int, int>` mapping from an integer name to a tid with a fixed size array in $O(1)$ time.

Tasks can register their own tid to a static integer name using `RegisterAs`. Tasks can also query the tid of a static name from the nameserver using `WhoIs`. Both are described in more detail below.

RegisterAs

```
int RegisterAs(int id);
```


RegisterAs queries the tid of the registered NameServer using **GetNS**, then calls the **Send** syscall with the nameserver tid and expects the NameServer to return a success or failure on registering the user task into the nameserver.

WhoIs

```
int WhoIs(int id);
```

WhoIs queries the tid of the registered NameServer using **GetNS**, and calls the **Send** syscall with the name server tid and expects the NameServer to return the associated tid to the name.

IRQ Context Switch

Kernel To User Task

When activating a user task, the kernel now checks the TaskDescriptor interrupt flag to determine if it is reloading a SWI or a hardware interrupted task. When reloading an IRQ, all registers are restored and **SUBS pc, lr, #4** is called.

User Task to Kernel

The user state is first copied onto the irq stack and then migrated to the user stack. The kernel state is then restored, and the SPSR is read from **irq_mode**. A special flag is set in the user's TaskDescriptor to note that it has hardware interrupted and should do a full restore (reload r0).

Interrupt Handling

Interrupt Handler

The Interrupt Handler supports the syscall **AwaitEvent** which user tasks can invoke to block waiting for an interrupt and be awoken from the Interrupt Matrix. The kernel will run the **event_register** command to register the task on the interrupt specified by the user task's r0 (subsequently saved to the user stack after context switch).

When a hardware interrupt occurs, the kernel enters via **IRQ_mode**, notes itself as a hardware interrupt and then runs **event_wake** which does a lookup in the Interrupt Matrix and wakes up all tasks blocking on the highest priority, non-empty interrupt. Any active interrupts with higher priority (which have no waiting tasks) will also be cleared to avoid multiple context switch overhead.

Interrupt initialization and cleanup are handled at the beginning and end of the kernel. Interrupt init creates sets up the Interrupt Matrix, sets the IRQ Mode stack base, and enables interrupts.

Interrupt cleanup disables all interrupts (so we don't have to reset the box after every session).

Note: In the spirit of microkernels, enabling/disabling specific interrupts may be moved to a user task instead.

```
void init_irq(interrupt_matrix *im );
void cleanup_irq();
void event_register(interrupt_matrix *im, TaskDescriptor *td);
void event_wake(interrupt_matrix *im);
```

Interrupt Matrix

The Interrupt Matrix is a data structure which holds all tasks blocked waiting on an interrupt. The matrix is a **struct** including circular buffers for each interrupt the kernel handles (improve as we go). Currently, the Interrupt Matrix only supports storing and loading tasks for the T3CUI interrupt.

Insertion and head deletion for the task both take $O(1)$ time. However, when waking all tasks for a corresponding interrupt requires $O(n)$ time, for n being the number of tasks waiting on the interrupt (rarely large).

```
void im_init(interrupt_matrix *im);
int im_push(interrupt_matrix *im, TaskDescriptor *task, InterruptEvent ie);
TaskDescriptor *im_top(interrupt_matrix *im, InterruptEvent ie);
int im_pop(interrupt_matrix *im, InterruptEvent ie);
unsigned int im_eventsize(interrupt_matrix *im, InterruptEvent ie);
```

Clock Server

Event Types

- Delay(tid, ticks) - queues a task for the given number of ticks
- DelayUntil(tid, tick) - queues a task until the given tick
- Update() - tells the clock server to increase the tick count (meant to be sent from the clock server notifier)
- Halt() - tells the clock server to shut-down

Implementation

The clock server is designed to be as simple as possible to minimize the chance of bugs. Under the hood it is very simple. The clock server maintains an ordered

queue of task ids. The task ids are ordered based on the tick that the task is waiting for. At the head of the linked-list is the task waiting for the lowest tick. We prioritized the **pop** operation to be as fast as possible making the trade-off of having a more expensive **insert** operation.

Insertion to the clock server queue occurs in $O(n)$ where n is the number of elements in the queue.

Popping the next ready task id off of the queue is $O(1)$. Since the queue is ordered no other adjustment has to occur when popping a task id.

The clock server itself is a typical send/receive server which loops infinitely on a **Request** and handles the different events.

When the clock server receives an **Update** event it updates the tick count and checks the queue for ready elements. Currently the clock server will pop off up to `CS_PROCESS_NUM == 5` tasks per **Update**.

See `src/lib/clockserver_queue.c` `src/user/clockserver.c` for implementation details.

Clock Server Notifier

In order to notify the clock server of when an interrupt occurs, a separate task is used to handle the interrupt and send a request to the clock server. The clock server notifier calls **AwaitEvent** in an infinite loop and then **Sends** to the clock server an **Update** request.

Syscalls

The kernel supports the following syscalls:

- **Assert**: Invoked via **assert** provides a method of testing in tasks
- **Create**: Creates another task to be put on the kernel's task schedule
- **GetTid**: Get the task's tid
- **GetParentTid**: Get the parent's task tid
- **Pass**: Give control away
- **Exit**: Become a zombie
- **Send**: sends a message to a tid
- **Receive**: receives a message from another task
- **Reply**: replies to a sender with a result
- **RegisterNS**: Registers a user task as the nameserver
- **GetNS**: Returns the tid of the current nameserver

Output

Raw Output (Snippet)

```
t6,d10,i1
t6,d10,i2
t7,d23,i1
t6,d10,i3
t8,d33,i1
t6,d10,i4
t7,d23,i2
t6,d10,i5
t6,d10,i6
t8,d33,i2
t7,d23,i3
t6,d10,i7
t9,d71,i1
t6,d10,i8
t6,d10,i9
t7,d23,i4
t8,d33,i3
t6,d10,i10
t6,d10,i11
t7,d23,i5
t6,d10,i12
t6,d10,i13
t8,d33,i4
t7,d23,i6
t6,d10,i14
t9,d71,i2
t6,d10,i15
t6,d10,i16
t7,d23,i7
t8,d33,i5
t6,d10,i17
t6,d10,i18
t7,d23,i8
t6,d10,i19
t8,d33,i6
t6,d10,i20
t7,d23,i9
t9,d71,i3
FINAL METRICS
=====
Idle ran for: 211 of 213 ticks
```

```

Percentage Idle: 99%
Worst Running Session: 0.99 tick
Best Running Session: 1.0 tick

```

Note: without -O2 and caching the final metrics become:

```

FINAL METRICS
=====
Idle ran for: 184 of 214 ticks
Percentage Idle: 85%
Worst Running Session: 0.69 tick
Best Running Session: 1.0 tick

```

Explanation

User Tasks Output

First, you may notice the very terse output formatting. This is because having a longer print statement actually delays tasks enough to mess with the ordering.

In our kernel, the larger the priority number, the more priority the task has. 31 is the highest priority and 0 is the lowest. The tasks are labeled **t6** through **t9**. **t6** has priority 6 (highest), **t7** has priority 5, **t8** has priority 4 and **t9** has priority 3.

To understand the correct ordering of the output we ran through the test case by hand. Consider the following timeline which demonstrates the ordering of events worked out by hand, assuming the amount of time it takes between a task resuming, printing and the next task delaying is less than a tick.

time (ms)	event	output
0	t6 unblocked, delays for 100ms	
0	t7 unblocked, delays for 230ms	
0	t8 unblocked, delays for 330ms	
0	t9 unblocked, delays for 710ms	
100	t6 resumes, prints, delays 100ms	t6,d10,i1
200	t6 resumes, prints, delays 100ms	t6,d10,i2
230	t7 resumes, prints, delays 230ms	t7,d23,i1
300	t6 resumes, prints, delays 100ms	t6,d10,i3
330	t8 resumes, prints, delays 330ms	t8,d33,i1
400	t6 resumes, prints, delays 100ms	t6,d10,i4
460	t7 resumes, prints, delays 230ms	t7,d23,i2
500	t6 resumes, prints, delays 100ms	t6,d10,i5
600	t6 resumes, prints, delays 100ms	t6,d10,i6
660	t8 resumes, prints, delays 330ms	t8,d33,i2
690	t7 resumes, prints, delays 230ms	t7,d23,i3

time (ms)	event	output
700	t6 resumes, prints, delays 100ms	t6,d10,i7
710	t9 resumes, prints, delays 710ms	t9,d71,i1
800	t6 resumes, prints, delays 100ms	t6,d10,i8
900	t6 resumes, prints, delays 100ms	t6,d10,i9
920	t7 resumes, prints, delays 230ms	t7,d23,i4
990	t8 resumes, prints, delays 330ms	t8,d33,i3
1000	t6 resumes, prints, delays 100ms	t6,d10,i10
1100	t6 resumes, prints, delays 100ms	t6,d10,i11
1150	t7 resumes, prints, delays 230ms	t7,d23,i5
1200	t6 resumes, prints, delays 100ms	t6,d10,i12
1300	t6 resumes, prints, delays 100ms	t6,d10,i13
1320	t8 resumes, prints, delays 330ms	t8,d33,i4
1380	t7 resumes, prints, delays 230ms	t7,d23,i6
1400	t6 resumes, prints, delays 100ms	t6,d10,i14
1420	t9 resumes, prints, delays 710ms	t9,d71,i2
1500	t6 resumes, prints, delays 100ms	t6,d10,i15
1600	t6 resumes, prints, delays 100ms	t6,d10,i16
1610	t7 resumes, prints, delays 230ms	t7,d23,i7
1650	t8 resumes, prints, delays 330ms	t8,d33,i5
1700	t6 resumes, prints, delays 100ms	t6,d10,i17
1800	t6 resumes, prints, delays 100ms	t6,d10,i18
1840	t7 resumes, prints, delays 230ms	t7,d23,i8
1900	t6 resumes, prints, delays 100ms	t6,d10,i19
1980	t8 resumes, prints, delays 330ms	t8,d33,i6
2000	t6 resumes, prints, delays 100ms	t6,d10,i20
2070	t7 resumes, prints, delays 230ms	t7,d23,i9
2130	t9 resumes, prints, delays 710ms	t9,d71,i3

With the assumption that the time between a task resuming, printing and delaying again is less than a tick, the above ordering holds. We noticed that increasing the size of the print message caused some lines to be out of order. Namely, `t7,d23,i3` and `t6,d10,i7` were out of order when we had a longer print message for each line.

As you can see the output of the kernel matches the hand example.

Metrics Output

IdleTask

The IdleTask serves the purpose of computing kernel metrics as well as a keep alive for hardware interrupts to occur.

The IdleTask measures its running time using the T3Timer and the ClockServer. Every busy loop iteration, the IdleTask compares its last checked time against the ClockServer to the current. If the tick values differ, then a hardware interrupt must have occurred.

The IdleTask on every busy loop iteration performs a delta on the T3Timer. If we've yet to be interrupted, sum the delta to the measured session. Once an interrupt has occurred, sum the remaining delta (prev to 0) to session and sum the session to total runtime. Once an interrupt occurs, the IdleTask asks the IdleTaskInterface for permission to exit.

This method provides a finer granularity than ClockServer ticks as the IdleTask will context switch in and out multiple times in one hardware interrupt. The IdleTask has the ability to measure fractions of a tick using the 502KHz clock (5020 cycles per tick).

Limitation: There will be a margin of error if the IdleTask is interrupted between queries to the clock server. This will result in the IdleTask thinking it has ran longer than it actually has. The margin of error will only ever be a fraction of a tick, and corrects itself when the IdleTask realizes the interrupt has occurred.

Note: Running time metrics may be moved to the kernel in the future to provide a more accurate and flexible measurement.

When IdleTask exits, it creates a ClockServerStop and NameServerStop task to stop the ClockServer and NameServer respectively. The kernel will then gracefully shutdown as no tasks exist.

IdleTaskInterface

The IdleTaskInterface serves as a gateway between the state of user tasks and the IdleTask. Since the IdleTask cannot block (unless for a short query), the IdleTaskInterface acts as a server listening to events happening in user land.

The first user task generates the IdleTaskInterface and sends a message about the number of client tasks which should run.

The IdleTask will periodically query the IdleTaskInterface (once every irq) for a request to exit. The IdleTaskInterface will reply `K3_NOT_FINISHED` unless all client tasks have finished.

When client tasks complete, they notify the IdleTaskInterface of their exit. Once all client tasks complete, IdleTaskInterface will respond to the next IdleTask query with `K3_FINISHED`.

Source Code and Hashes

Source is located at <https://git.uwaterloo.ca/bkcs452/kernel/tree/kernel3>.

- 3832d3ebce59f64f05aaa29ea99569cc ./include/asm/asm.h
- 9ca141fc00c63f532a54da40db8c5f98 ./include/types.h
- 9fecb4b5deb16f92c7ea7cbd807c567e ./include/lib/clockserver_queue.h
- 27539e09c4cd89fc74f465a7b5c0e5e7 ./include/lib/va_arg.h
- 1665ad3c8f2988556b8d1381a293da88 ./include/lib/circularbuffer.h
- 607978647233ff95b922813e963f2bba ./include/ts7200.h
- cba9ecc3d9f7d06716c6416a90ebf2eb ./include/kernel/kernel.h
- de694d347f568a8d62cf2c5f4e815f3d ./include/kernel/handlers/msg.h
- 6d8242f2b8dcd143ab9335157da714ff ./include/kernel/handlers/nameserver.h
- 63072912747c71f41b867ad163b29a16 ./include/kernel/handlers/interrupt.h
- 75ba899de21e5a56cd03e5463b55454d ./include/kernel/kernel_task.h
- fb0dbad170938a28ca9c81d4fe3de845 ./include/kernel/syscalls.h
- b1f8047ff7f478cf19f5ad96971112aa9 ./include/debug/debug.h
- 6b8865733725756f1085e88f04f90a23 ./include/stdlib.h
- 17db0108f18acb4142b91a890214d4f2 ./include/user/test/k2_task.h
- a9cab74b7721adb46af0ddf32a2d7841 ./include/user/test/messaging.h
- 62dd92891677663ef7950e33f853f902 ./include/user/test/k1_task.h
- 63368c63b6c74973540f5337d40f6818 ./include/user/test/test_task.h
- 76fbcc66e11439d9b51e66121d4efa3a ./include/user/test/k2_metric.h
- 8b67f7312748456c9d148ad0bfa76d56 ./include/user/test/k3_task.h
- d7eb03b532bea124e499219263457b33 ./include/user/nameserver.h
- 9c0b407c3a06cc21253d93d5b5a263f4 ./include/user/clockserver.h
- 819537e82704c7fde1fccc0a127df1da ./include/io/bwio.h
- e7e2dd4c821d2110beac97cf286c0d73 ./include/io/io.h
- 38572de424efd3e363c6ad7a15928fa4 ./include/test/priority_queue.h
- 2e764754296ba831c89b7e03929de399 ./include/test/clockserver_queue.h
- 660038d9adc85b7933a6e8444dbfe997 ./include/test/test.h
- 2d84854ea39152a60b5d286e92ead0e8 ./include/test/circularbuffer.h
- 785d5a23aea9c2b10b707a559ed67e49 ./include/test/nameserver.h
- 3a032233a9e49684452f8454fce84c0c ./include/test/task_queue.h
- 5ed44bb6d05497a403ca33a791973c26 ./include/defines.h
- 3774a5c58cb4a0d95fd75338d9eefe9d ./include/system.h
- bab426b2c45207b0d1743e4e5dc759e4 ./include/task/interrupt_matrix.h
- d81ff69cda76f0fd9e82496dee58d35b ./include/task/priority_queue.h
- 0317a55d6b4f27a38b7c7e84d8c0aa9c ./include/task/task.h
- 3ba7bf8bc6390be93bcd1412808e92e9 ./include/task/task_queue.h
- feb15bff0375434e45062dbb82aa7a47 ./src/lib/clockserver_queue.c
- 0b3a6bd753bec252fd0ab8e7a0047f2b ./src/lib/circularbuffer.c
- d31c942346b1ea403e8d7688fe64b7dc ./src/kernel/kernel.c
- c1ec17cac8dbb5c38d8dd6d885d796dc ./src/kernel/handlers/msg.c
- 42e2896ddc1f3d820d42f262b5c6cfe3 ./src/kernel/handlers/interrupt.c
- 08cee571550f520b5ffd8967f62a8519 ./src/kernel/handlers/nameserver.c

- 2dba7ee4eba61d3ed989f090d9f8bfd4 ./src/kernel/kernel_task.c
- d96b9581953c326e6597410935f6313e ./src/kernel/syscalls.c
- 87302e3265397e8151994039b7a8c5c4 ./src/kernel/system.c
- 2c68ab67bba20d39641686afd55d1ff5 ./src/user/clockserver.c
- cda628304ed006bd9b52660913eb8b89 ./src/user/nameserver.c
- 7def167f954c85f76a8bd4ea4d80f6ad ./src/user/test/k2_metric.c
- 706088ec9b7330a3c4904dffa46c2fe8 ./src/user/test/k1_task.c
- b75b9a70bb199ce1b73105eb80ec081e ./src/user/test/test_task.c
- c303568e6c8f22b805e0a5c37e1dbf20 ./src/user/test/k3_task.c
- 97a6d138065e14cc5da3f114d824d463 ./src/user/test/k2_task.c
- ed1c5d3d11fafc6001d7403e0da8ed04 ./src/user/test/messaging.c
- ca0df9fc349ef064b27bc743a5397b3c ./src/io/bwio.c
- 59fd88c4ba82dfada61dae85260375ef ./src/test/circularbuffer/circularbuffer.c
- 4597316f7dce4804c65ae5bf95b39cfd ./src/test/clockserver_queue.c
- 14d6634474785af66d97b133071e5e72 ./src/test/test.c
- 7ad05d592124cd75e534b949cd04dc97 ./src/test/nameserver/nameserver.c
- 3df71775258a71dc7ce9a775e2993263 ./src/test/task/priority_queue.c
- 7bc01453e2a66825b5d984a4216e3574 ./src/test/task/task_queue.c
- 49ae81eda219a859562be10486ba854e ./src/task/priority_queue.c
- 1ed3532066c99ed5f28a19a1d13e6eff ./src/task/interrupt_matrix.c
- 5dc340880be1efbc58b42adca2a3e701 ./src/task/task.c
- e0a93e5f5b3d54fde762c8f94c6cbb26 ./src/task/task_queue.c