NOTES FOR CS452
KERNEL DESCRIPTION

BILL COWAN
UNIVERSITY OF WATERLOO

A. PROGRAMMER'S MODEL

A.I. TASK CREATION

A.I.I. *Create*

Name. `Create` - instantiate a task.

Synopsis. `int Create( int priority, void (*code) ( ) )`

Description. `Create` allocates and initializes a task descriptor, using the given priority, and the given function pointer as a pointer to the entry point of executable code, essentially a function with no arguments and no return value. When `Create` returns the task descriptor has all the state needed to run the task, the task's stack has been suitably initialized, and the task has been entered into its ready queue so that it will run the next time it is scheduled.

Returns.

- `tid` – the positive integer task id of the newly created task. The task id must be unique, in the sense that no task has, will have or has had the same task id.
- -1 – if the priority is invalid.
- -2 – if the kernel is out of task descriptors.

You might want to do some rough tests to ensure that the function pointer argument is valid: it's not obvious to me which of many possible tests would be most useful.

A.I.II. *-MyTid*

Name. `MyTid` - find my task id.

Synopsis. `int MyTid( )`

Description. `MyTid` returns the task id of the calling task.

Returns.

- `tid` – the positive integer task id of the task that calls it.
- Errors should be impossible!

A.I.III. *MyParentTid*

Name. `MyParentTid` - find the task id of the task that created the running task.

Synopsis. `int MyParentTid( )`

Description. `MyParentTid` returns the task id of the task that created the calling task.

This will be problematic only if the task has exited or been destroyed, in which case the return value is implementation-dependent.

Returns.

- `tid` – the task id of the task that created the calling task.
- The return value is implementation-dependent if the parent has exited, has been destroyed, or is in the process of being destroyed.

A.1.IV. *Pass*

Name. `Pass` - cease execution, remaining ready to run.

Synopsis. `void Pass( )`

Description. `Pass` causes a task to stop executing. The task is moved to the end of its priority queue, and will resume executing when next scheduled.

A.1.V. *Exit*

Name. `Exit` - terminate execution forever.

Synopsis. `void Exit( )`

Description. `Exit` causes a task to cease execution permanently. It is removed from all priority queues, send queues, receive queues and awaitEvent queues. Resources owned by the task, primarily its memory and task descriptor are not reclaimed.

Returns. `Exit` does not return. If a point occurs where all tasks have exited the kernel should return cleanly to RedBoot.

A.1.VI. *Destroy*

Please see the separate document for Destroy. Re-using resources is complicated.

A.2. INTER-TASK COMMUNICATION

A.2.I. *Send*

Name. `Send` - send a message.

Synopsis. `int Send( int Tid, char *msg, int msglen, char *reply, int replylen )`

Description. `Send` sends a message to another task and receives a reply. The message, in a buffer in the sending task's address space is copied to the address space of the task to which it is sent by the kernel. `Send` supplies a buffer into which the reply is to be copied, and the size of the buffer so that the kernel can detect overflow. When `Send` returns without error it is guaranteed that the message has been received, and that a reply has been sent, not necessarily by the same task. If either the message or the reply is a string it is necessary that the length should include the terminating null.

The kernel will not overflow the reply buffer. The caller is expected to compare the return value to the size of the reply buffer. If part of the reply is missing the return value will exceed the size of the supplied reply buffer.

There is no guarantee that `Send` will return. If, for example, the task to which the message is directed never calls `Receive`, `Send` never returns and the sending task remains blocked forever.

`Send` has a passing resemblance, and no more, to remote procedure call.

Returns.

- The size of the message supplied by the replying task.
- -1 – if the task id is impossible.
- -2 – if the task id is not an existing task.
- -3 – if the send-receive-reply transaction is incomplete.

A.2.II. *Receive*

Name. `Receive` - receive a message.

Synopsis. `int Receive( int *tid, char *msg, int msglen )`

Description. `Receive` blocks until a message is sent to the caller, then returns with the message in its message buffer and `tid` set to the task id of the task that sent the message. Messages sent before `Receive` is called are retained in a send queue, from which they are received in first-come, first-served order.

The argument `msg` must point to a buffer at least as large as `msglen`. If the size of the message received exceeds `msglen`, no overflow occurs and the buffer will contain the first `msglen` characters of the message sent.

The caller is expected to compare the return value, which contains the size of the message that was sent, to determine whether or not the message is complete, and to act accordingly.

Returns.

- The size of the message sent.

A.2.III. *Reply*

Name. `Reply` - reply to a message.

Synopsis. `int Reply( int tid, char *reply, int replylen )`

Description. `Reply` sends a reply to a task that previously sent a message. When it returns without error, the reply has been copied into the sender's address space. The calling task and the sender return at the same logical time, so whichever is of higher priority runs first. If they are of the same priority the sender runs first.

Returns.

- 0 – if the reply succeeds.
- -1 – if the task id is not a possible task id.
- -2 – if the task id is not an existing task.
- -3 – if the task is not reply blocked.

- -4 – if there was insufficient space for the entire reply in the sender's reply buffer.

A.3. Name Server

A.3.i. *RegisterAs*

Name. `RegisterAs` - register a name with the name server.

Synopsis. `int RegisterAs( char *name )`

Description. `RegisterAs` registers the task id of the caller under the given name.

On return without error it is guaranteed that all `WhoIs` calls by any task will return the task id of the caller until the registration is overwritten.

If another task has already registered with the given name its registration is overwritten.

A single task may register under several different names, but each name is assigned to a single task.

`RegisterAs` is actually a wrapper covering a send to the name server.

Returns.

- 0 – success.
- -1 – if the nameserver task id inside the wrapper is invalid.
- -2 – if the nameserver task id inside the wrapper is not the name server.

A.3.ii. *WhoIs*

Name. `WhoIs` - query the nameserver.

Synopsis. `int WhoIs( char *name )`

Description. `WhoIs` asks the nameserver for the task id of the task that is registered under the given name.

Whether `WhoIs` blocks waiting for a registration or returns with an error if no task is registered under the given name is implementation-dependent.

There is guaranteed to be a unique task id associated with each registered name, but the registered task may change at any time after a call to `WhoIs`.

`WhoIs` is actually a wrapper covering a send to the nameserver.

Returns.

- `tid` – the task id of the registered task.
- -1 – if the nameserver task id inside the wrapper is invalid.
- -2 – if the nameserver task id inside the wrapper is not the nameserver.

A.4. Interrupt Processing

A.4.i. *AwaitEvent*

Name. `AwaitEvent` - wait for an external event.

Synopsis. `AwaitEvent` gives you more freedom that you have had up until now. There are two possible signatures

`int AwaitEvent( int eventid, char *event, int eventlen )`

or

`int AwaitEvent( int eventid )`

and several choices for implementation, given below.

Description. `AwaitEvent` blocks until the event identified by eventid occurs then returns.

The following details are implementation-dependent.

- Whether or not the kernel collects volatile data and re-enables the interrupt.
- Whether volatile data is returned as a positive integer in the return value, or in the event buffer, or not returned at all.
- Whether or not interrupts are enabled when AwaitEvent returns.
- Whether or not to allow more than one task to block on a single event.

The trade-offs associated with these implementation decisions will be discussed in class, and see below §B.4.

Returns.

- volatile data – in the form of a positive integer.
- 0 – volatile data is in the event buffer.
- -1 – invalid event.
- -2 – corrupted volatile data. Error indication in the event buffer.
- -3 – volatile data must be collected and interrupts re-enabled in the caller.

A.5. Clock Server

A.5.i. *Delay*

Name. `Delay` - wait for a given amount of time.

Synopsis. `int Delay( int ticks )`

Description. `Delay` returns after the given number of ticks has elapsed. How long after is not guaranteed because the caller may have to wait on higher priority tasks.

`Delay` is (almost) identical to `Pass` if `ticks` is zero or negative.

The size of a tick is normally application dependent. In cs452 this term it is 10 milliseconds.

`Delay` is actually a wrapper for a send to the clock server.

Returns.

- 0 – success.
- -1 – if the clock server task id inside the wrapper is invalid.
- -2 – if the clock server task id inside the wrapper is not the id of the clock server.

A.5.ii. *Time*

Name. `Time` - give the time since clock server start up.

Synopsis. `int Time( )`

Description. `Time` returns the number of ticks since the clock server was created and initialized.

With a 10 millisecond tick and a 32-bit int there should be neither wraparound nor negative time.

`Time` is actually a wrapper for a send to the clock server.

Returns.

- non-negative integer – time in ticks since the clock server initialized.
- -1 – if the clock server task id inside the wrapper is invalid.
- -2 – if the clock server task id inside the wrapper is not the task id of the clock server.

A.5.III. *DelayUntil*

Name. `DelayUntil` - wait until a time.

Synopsis. `int DelayUntil( int ticks )`

Description. `Delay` returns when the time since clock server initialization is greater than the given number of ticks. How long after is not guaranteed because the caller may have to wait on higher priority tasks.

`DelayUntil ( Time( ) + ticks )` is (almost) identical to `Delay ( ticks )`.

The size of a tick is normally application dependent. In cs452 this term it is 50 milliseconds.

`DelayUntil` is actually a wrapper for a send to the clock server.

Returns.

- 0 – success.
- -1 – if the clock server task id inside the wrapper is invalid.
- -2 – if the clock server task id inside the wrapper is not the clock server.

A.6. INPUT/OUTPUT

A.6.I. *Getc*

Name. `Getc` - get a character from a UART.

Synopsis. `int Getc( int channel )`

Description. `Getc` returns first unreturned character from the given UART.

How transmission errors are handled is implementation-dependent.

`Getc` is actually a wrapper for a send to the serial server.

Returns.

- character – success.
- -1 – if the serial server task id inside the wrapper is invalid.
- -2 – if the serial server task id inside the wrapper is not the serial server.

A.6.II. *Putc*

Name. `Putc` - send a character from the given UART.

Synopsis. `int Putc( int channel, char ch )`

Description. `Putc` queues the given character for transmission by the given UART. On return the only guarantee is that the character has been queued. Whether it has been transmitted or received is not guaranteed.

How configuration errors are handled is implementation-dependent.

`Putc` is actually a wrapper for a send to the serial server.

Returns.

- 0 – success.
- -1 – if the serial server task id inside the wrapper is invalid.
- -2 – if the serial server task id inside the wrapper is not the serial server.

B. Algorithms and Data Structures

All algorithms must be constant-time, in the sense that you can bound them above at a reasonable level. 'Reasonable' means on a time scale appropriate for a train application.

All data must be either static or on the stack. There is no heap; there is no dynamic memory allocation, at least as you know it in C, C++ or Java running under Unix.

B.I. Task Descriptors

B.I.I. *Basics*

Each task has a task descriptor (`TD`), which is the most important data structure in the kernel. The `TD`s should be local to the kernel, that is, on the kernel stack, allocated when the kernel begins executing. Each `TD` must include

1. a task identifier unique to this instance of the task,
2. the task's state,
3. the task's priority,
4. the task identifier of the task's parent, the task that created this task,
5. a stack pointer pointing to this task's private memory,
6. this task's saved program status register (`SPSR`), and
7. the task's return value, which is to be returned to the task when it next executes.

The `TD` may include other fields. For example, a popular priority queue implementation is a doubly-linked circular list, using pointers to the `TD`s of tasks ahead and behind the task in the queue that are part of the `TD`.

B.I.II. *Comments*

1. When `Destroy` is not implemented the task id can be its array index because `TD`s are not re-used. When `Destroy` is implemented a better task id model is needed.
2. A task can be in one of the following states.
   i. Ready. The task is ready to be activated.

   ii. Active. The task that has just run, is running, or is about to run. Scheduling, which happens near the end of kernel processing, changes the active task. On a single processor only one task can be active at a time.
   iii. Zombie. The task has exited.
   iv. Send-blocked. The task has executed `Receive`, and is waiting for a message to be sent to it.
   v. Receive-blocked. The task has executed `Send`, and is waiting for the messagre to be received.
   vi. Reply-blocked. The task has executed `Send` and its message has been received, but it has not received a reply.
   vii. Event-blocked. The task has executed `AwaitEvent`, but the event on which it is waiting has not occurred.

   The first three of these states are needed for task creation; the next three are needed for message passing; and the seventh is needed for hardware interrupts.
3. The parent is the active task when a task is being created. This entails that the variable storing the active task is written only by the scheduler.
4. A task has memory reserved for it when it is created, which it uses for its stack. The values of its registers are placed on the stack when it is not executing.
5. Each task has, when it is running, a program status register (CPSR), which is saved when it is interrupted and re-installed when the task is next activated.
6. Tasks usually enter the kernel with requests for service. Many tasks must be provided with return values that indicate the result of the request. Because a task may not be rescheduled immediately the return value must be saved.

B.2. Scheduling

Scheduling is done using priorities: non-blocked higher priority tasks are guaranteed to execute before lower priority ones.

1. The number of priorities is implementation-dependent.
2. There can be more than one task at any priority. As tasks become ready to run, they are put on the end of their ready queue. The next task to run is always the highest priority task that is ready. If more than one task is ready at the highest priority, then the one at the head of the ready queue runs.
3. A task instance may not be at more than one priority.

Occasionally, or even frequently, the ready queues are empty with one or more tasks Event-Blocked. In this case there are two things you might do.

1. You can have an idle task at the lowest priority, which does anything you want while it waits to be interrupted.
2. You can put the CPU into the halt state.

The second consumes less power, but power consumption is unimportant in cs452.

During the first two parts of kernel development the kernel should exit cleanly to RedBoot when there are no tasks in the ready queues. Once interrupts are introduced the kernal should exit cleanly to RedBoot when there are no Event-blocked tasks and no tasks in the ready queues. Kernels do not normally exit.

B.3. Context Switching

Those context switches into the kernel that occur when a running task requests a kernel service MUST be implemented using the ARM `swi` instruction. Context switches generated by interrupts are defined by the architecture.

The ARM architecture offers a variety of methods for exiting the kernel into a task. Choose whichever suits you.

B.4. Interrupts

The kernel should run with interrupts disabled. Thus, as little as possible should be done by the kernel because the time of the slowest kernel primitive must be added to the worst case response time of every action.

AwaitEvent requires a table of event ids, which is, in essence a catalogue of hardware events to which the kernel and servers are able to provide a response. This is an awkward intrusion of hardware configuration into an otherwise clean operating system abstraction. There are many different ways of handling this issue within operating system design. Arguments between proponents of different approaches are, for the most part, religious, which means that although the arguments are heated and often personal, they only rarely change the minds of the arguers. Fortunately for you, the hardware environment in which you work is circumscribed and well-defined, so that you can create a small registry of events, provide a reasonably consistent set of responses, and not think too much about the general problem. But we will discuss the general problem in class, just because it is quite pervasive, affecting many different areas of computer science.

If you happen to have a really good solution, and can persuade the world to take it up you are well on your way to your first billion.

B.5. Message Formats

Type checking of messages is not required, even though it would conceivably be very useful.

In most kernels and applications few, if any, messages are character strings; almost all messages contain the contents of a `struct`. Type checking might be handy, but can only be performed at run time because there is no restriction on which task can send to which other task. You can get pretty close to run-time type checking by giving every structure a field, the value of which is its type. What you do when you discover a mismatch is the hard part. Can you recover at run-time? Or do you need to stop and reprogram.

You will, no doubt, find that early design of a set of message structures, combined with discipline in your task structure, is important in defending you from bugs that can take a long time to find.