CENG 448 — Real Time Operating Systems

Semaphore and Mutex

Dr. Larry D. Pyeatt

South Dakota School of Mines and Technology

Semaphores and Mutexes

Provide a mechanism for synchronizing execution and coordinating mutually exclusive access to shared resources.

Mutex is a special type of Semaphore.

Semaphore

A kernel object (data structure) that one or more threads can acquire or release for the purpose of synchronization or mutual exclusion.

- Race conditions,
- Producer-consumer problem.

Typical implementation

Semaphores allow tasks to request a “token” to use a certain resource. For example, we may have a small number of DMA channels available, or we may have a device that only one task can access at any given time.

- Semaphore control block,
- Counter,
- Semaphore task waiting list.

If the counter in a semaphore is initialized to 1, then it is a *binary* semaphore.

Otherwise, it is a *counting* semaphore.

Mutex

A binary semaphore is not truly a mutex: A mutex keeps track of which task currently holds it.

A mutex:

- can only be released by the task that acquired it,
- can automatically raise the priority of the task that is holding the mutex, if a higher priority task requests the mutex,
- can be freed automatically when the task holding it terminates,
- can be recursively locked (locked multiple times) by the task holding it, and
- can signal an error if the task trying to release it does not own it.

Mutex is used for *locking.* One resource.

Semaphore is used for *signaling.* Multple instances of a resource.

Typical Operations

Create:	create the semaphore and make it available.
Delete:	destroy the semaphore.
Acquire:	attempt to get a token and return a value indicating success/failure.
Release:	return a token.
Flush	unblock all tasks waiting on a semaphore (task synchronization, or *thread rendezvous*).
Show	return general information about the semaphore.
Show blocked	return list of blocked tasks.

Variants on the acquire function

Wait forever: The acquire function puts the task on the semaphore waiting list and stays there until it gets a token.

Wait with timeout: The acquire function puts the task on the semaphore waiting list, but when its timeout is reached, it goes back to ready/running, and the acquire function returns a value indicating failure.

Do not wait: The acquire function immediately returns a value indicating failure.

Wait and signal

Synchronization without data exchange.

1. A binary semaphore is initialized with a count of 0 (unavailable).
2. `Taska` has higher priority than `Taskz`, so it will run first. It makes a call to acquire the semaphore, and is blocked.
3. `Taskz` now gets to run. It does not acquire the semaphore, but at some point it releases the semaphore.
4. As soon as the semaphore is released, `Taska` will preempt `Taskz` and resume execution.
5. The semaphore was used to ensure that `Taskz` reached a specific point before `Taska` continued, even though `Taska` has higher priority.

Multiple task wait and signal

1. A binary semaphore is initialized with a count of 0 (unavailable).
2. `Taska`, `Taskb`, and `Taskc` all have higher priority than `Taskz`, so they will run first. They all make a call to acquire the semaphore, and are blocked.
3. `Taskz` now gets to run. It does not acquire the semaphore, but at performs some work that the other tasks need (initialization or providing some data) and performs a `flush` operation on the semaphore.
4. All three blocked tasks become ready/running. The highest priority task will preempt `Taskz` and resume execution.
5. All three tasks will be blocked inside the acquire function, which will return an implementation-dependent “return from flush” value when they are resumed.

Credit-tracking synchronization

The signaling task may run faster than the signaled task.

For example, one task (the producer) may receive data, process it, and hand it off to another task (the consumer) in chunks.

A counting semaphore can keep track of how many chunks are available for the consumer to process.

1. The semaphore is initialized to 0.
2. The (higher priority) producer runs and processes some chunks, releasing the semaphore after each chunk. Eventually, it blocks waiting for input, or for some other reason (the seaphore is at maximum count).
3. The consumer gets to run, acquires a token, processes a chunk, and repeats until it blocks.

Producer could be in an ISR. Semaphore tracks how many interrupts have occurred.

Single shared-resource-access synchronization

1. The semaphore is initialized to 1.
2. Any task wishing to use the resource must first acquire the semaphore.
3. It is better to use a mutex for this, because a mutex provides better error protection. (What if one of the tasks is buggy, and releases the semaphore without acquiring it?)

Example: *Serializing* access to a device, such as serial port.

Each single shared resource should be protected by a mutex.

Our system has two UARTS. Are they two interchangeable resources, or two single shared resources?
Simultaneous (concurrent) access vs serialized access.

Recursive shared-resource-access synchronization

Task calls a function, which calls another function, and all three need to access a shared resource.

But those functions may be used by other tasks that don't directly acquire the shared resource, so the functions themselves need to make sure that the resource is acquired.

This is a job for the recursive mutex.

A mutex tracks who owns it. A recursive mutex allows the owner to own it multiple times, but no other task can get it if it is owned at least once.

Multiple shared-resource-access synchronization

If there are multiple resources that are equivalent (e.g. DMA channels), then a counting semaphore can be used to allocate them.

Care must be taken to ensure that a task does not release a semaphore that it has not acquired.

A separate mutex can be assigned for each shared resource. This takes more mutexes, but can be safer.