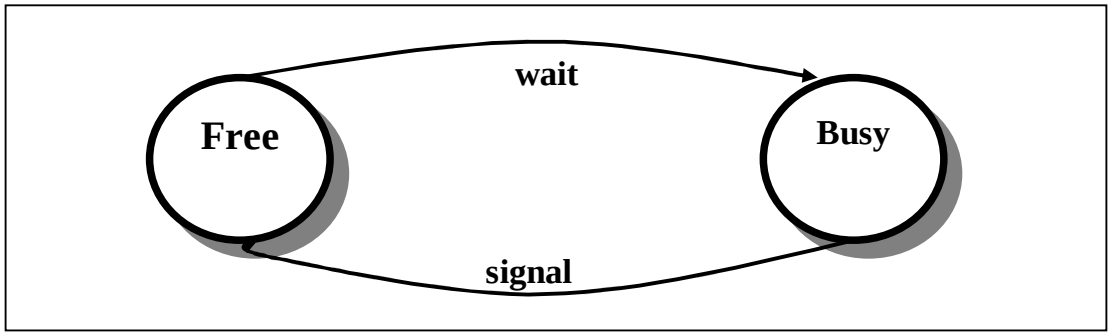
Exercise 1: A Mutex Semaphore

Write a process that will act as a binary semaphore providing mutual exclusion (mutex) for processes that want to share a resource. Model your process as a finite state machine with two states, busy and available. If a process tries to take the mutex (by calling **mutex:wait()**) when the process is in state busy, the function call should hang until the mutex becomes available (namely, the process holding the mutex calls **mutex:signal()**).

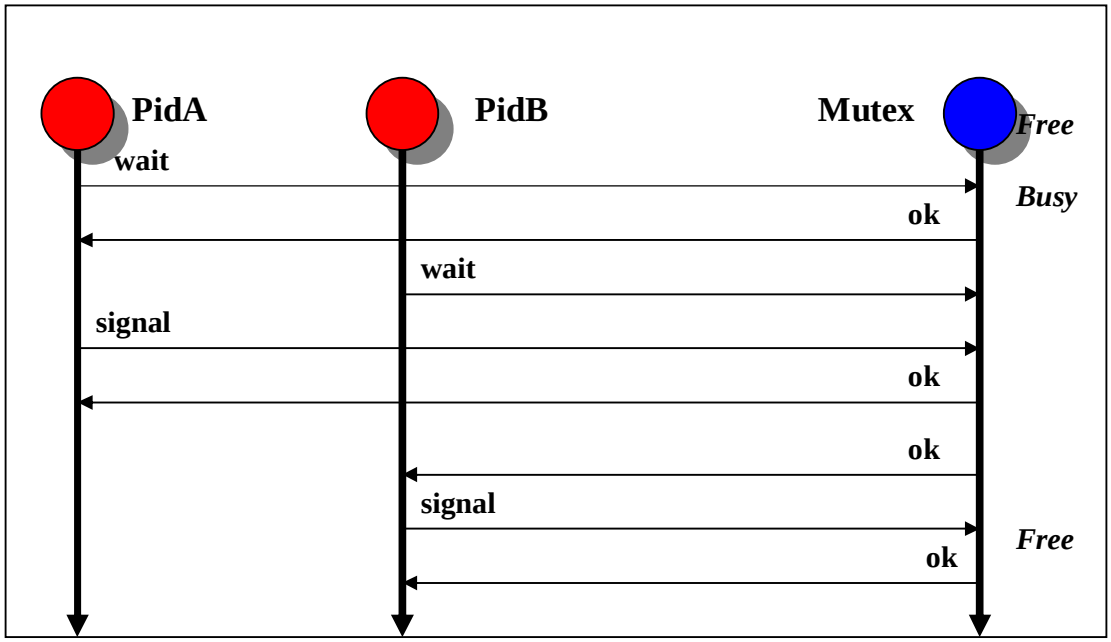
Interface

mutex:start() ⇒ ok.
mutex:wait() ⇒ ok.
mutex:signal() ⇒ ok.

Finite State Machine



Message Sequence Chart



Hint: The difference in the state of your FSM is which messages you handle in which state.

Exercise 2: A Reliable Mutex Semaphore

Your Mutex semaphore is unreliable. What happens if a process that currently holds the semaphore terminates prior to releasing it? Or what happens if a process waiting to execute is terminated due to an exit signal? By trapping exits and linking to the process that currently holds the semaphore, make your mutex semaphore reliable.

Hint: Use **catch link(Pid)** in case Pid terminated before its request was handled. This will result in an exit signal being sent to the process if you are trapping exits or {'EXIT', Reason} if you are not.