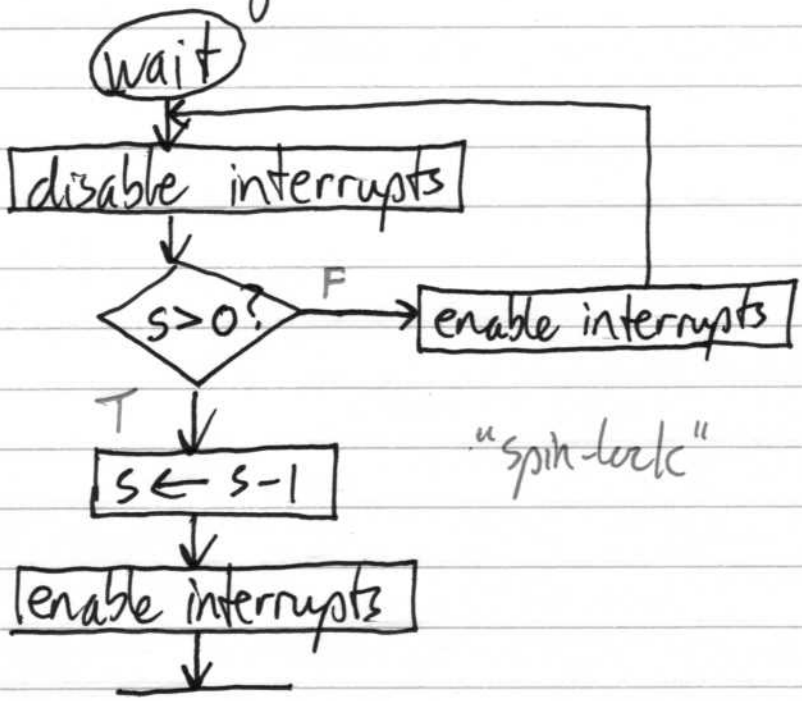
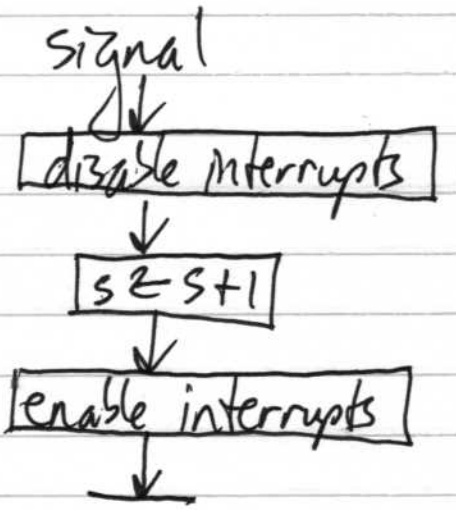
Synchronization

1) Semaphores [9.5]

- a semaphore is a counter with 3 functions
  1. init - initialize counter value S
  2. wait - tries to decrement counter; wait if S=0
  3. signal - increments counter



"spin-lock"



- read-modify-write sequences are protected by disabling interrupts
- the spin lock must enable interrupts each iteration to allow the SysTick interrupt so the scheduler can run (so other tasks can run)

4.1) Example Semaphore Implementation without RTX

```
typedef uint32_t  sem_t;

void init(sem_t *s, uint32_t count) {
    *s = count;
}

void wait(sem_t *s) {
    __disable_irq();                          ← CMSIS
    while(!(*s > 0)) {
        __enable_irq();  ← this gives scheduler a chance to run
        __disable_irq();
    }
    (*s)--;
    __enable_irq();
}

void signal(sem_t *s) {
    __disable_irq();
    (*s)++;
    __enable_irq();
}
```

2) Mutual Exclusion

- protect code that accesses shared data (critical code) by using a semaphore as a lock

- steps:
  1. initialize the semaphore lock to 1 (open)
  2. invoke wait() to acquire the lock before entering the critical section
  3. invoke signal() to release the lock when leaving the critical section

e.g.

```
sem_t lock;

int main(void) {
    init(&lock, 1);
    ... start task 1 and task 2
}

void task1(void) {
    ...
    wait(&lock);          // acquire
    - execute in critical section
    signal(&lock);        // release
    ...
}

void task2(void) {
    ...
    wait(&lock);
    - execute in critical section
    signal(&lock);
    ...
}
```

- only the lock "owner" should release it

3) Condition Variable

- one task signals another task that an event has occured; then the other task can proceed

- steps:
  1. initialize the condition variable semaphore to 0
  2. one task invokes wait() to wait for the event
  3. the other task (or ISR) invokes signal() to indicate that the event occured

e.g. `sem_t cond;` condition variable

```
int main(void) {
    init(&cond, 0);
    -start task1 and task2
}
```

```
void task1(void){
  // wait for event
  wait(& cond);
  -continue execution
  ...
}
```

```
void task2(void) {
  // detect event
  signal(& cond);
  -continue execution
  ...
}
```

4) Task Rendezvous

- synchronize two tasks to perform work at the same time

- steps:
  1. initialize two semaphores to 0
  2. each task signals the other task
  3. both tasks wait for the other's signal

e.g. `sem_t s1, s2;`

```
int main(void){
    init(&s1, 0);
    init(&s2, 0);
    - start task1 and task2
}
```

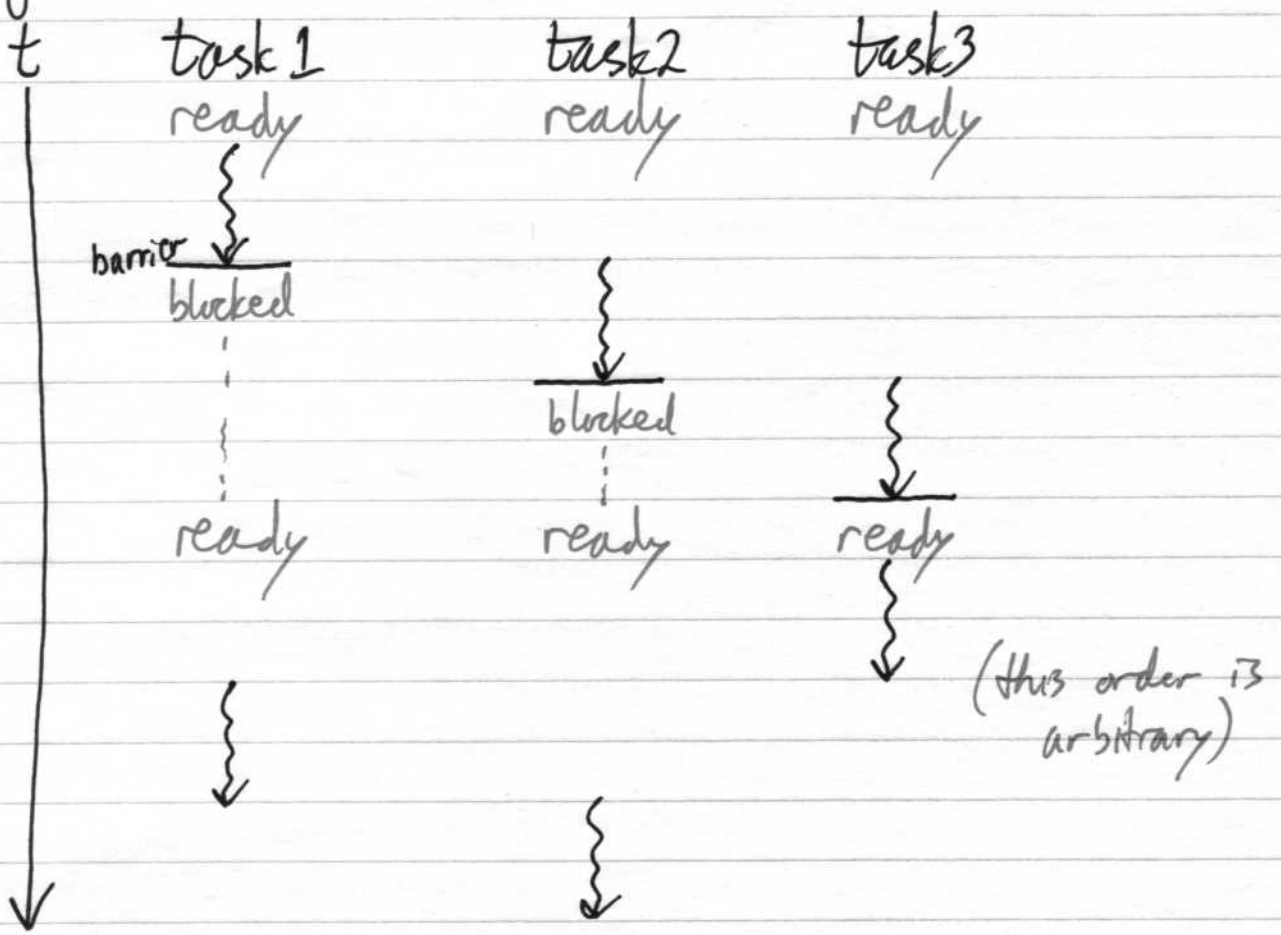
```
void task1(void){
    ...
    //rendezvous
    signal(&s2);
    wait(&s1);
    ...
}
```

```
void task2(void){
    ...
    //rendezvous
    signal(&s1);
    wait(&s2);
    ...
}
```

5) Barrier Synchronization

- all tasks must arrive at the barrier before any proceed past it
  i.e. it is an n-task rendezvous

eg. 3 task barrier



- we can try to solve with n semaphores

eg. task1 s1 task2 s2 task3 s3 (semaphores)

```
signal(&s2);
wait(&s1);
```

```
signal(&s3);
wait(&s2);
```

```
signal(&s1);
wait(&s3);
```

okay?
NO

- see further A3

- general solution [The Little Book of Semaphores, Downey]
  - 1 barrier semaphore
  - 1 mutex semaphore
  - 1 count variable

- steps:
  (1) initialize barrier = 0, mutex = 1, count = 0
  (barrier: make tasks wait; count: track arrivals; mutex: protect accesses to count)

  (2) each task:
  - acquire mutex
  - increment count
  - release mutex

  (3) last task signals the barrier

  (4) each task:
  - wait on barrier
  - signal barrier

  } "turnstile"

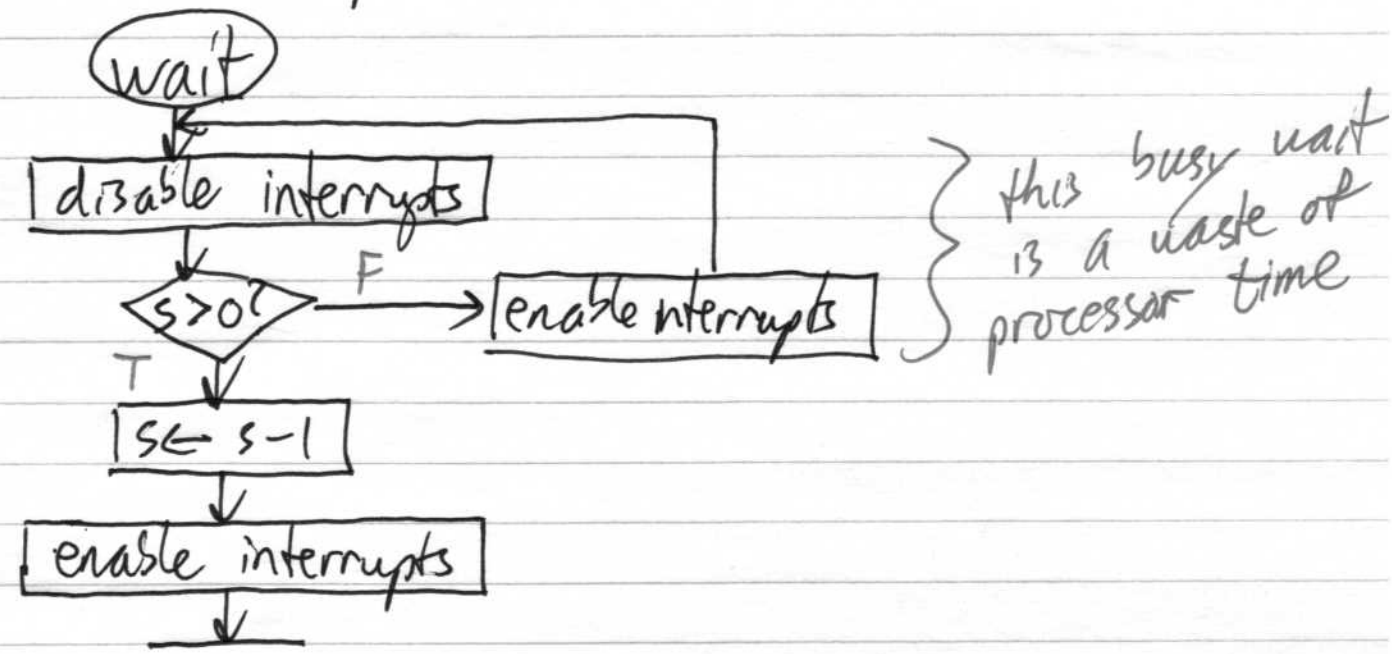
- implementation {

```
sem_t barrier, mutex;
uint32_t count;

void barrierInit(void) {
    init(&barrier, 0);
    init(&mutex, 1);
    count = 0;
}

void barrierSync(void) {
    wait(&mutex);
(critical section)  count++;
    signal(&mutex);

    if(count == n)
        signal(&barrier);

    wait(&barrier);
    signal(&barrier);
}

int main(void) {
    barrierInit();
    // start n tasks
}

void task(void) {
    ...
    barrierSync();
    ...
}
```

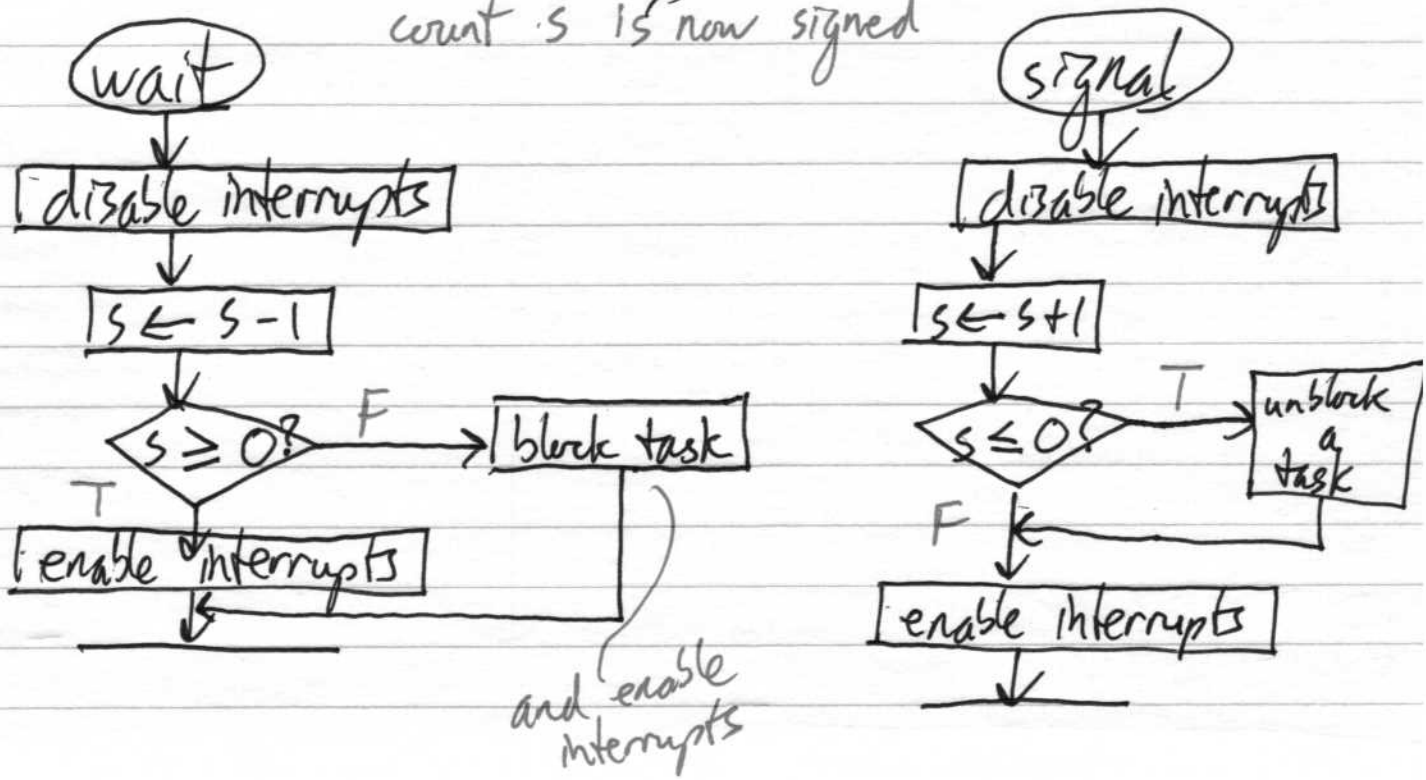
- value of the barrier semaphore will be in [1, n] after all have synced
- is this barrier re-usable?

6) Example Implementation of a Blocking Semaphore

- recall the "spin lock" in wait()

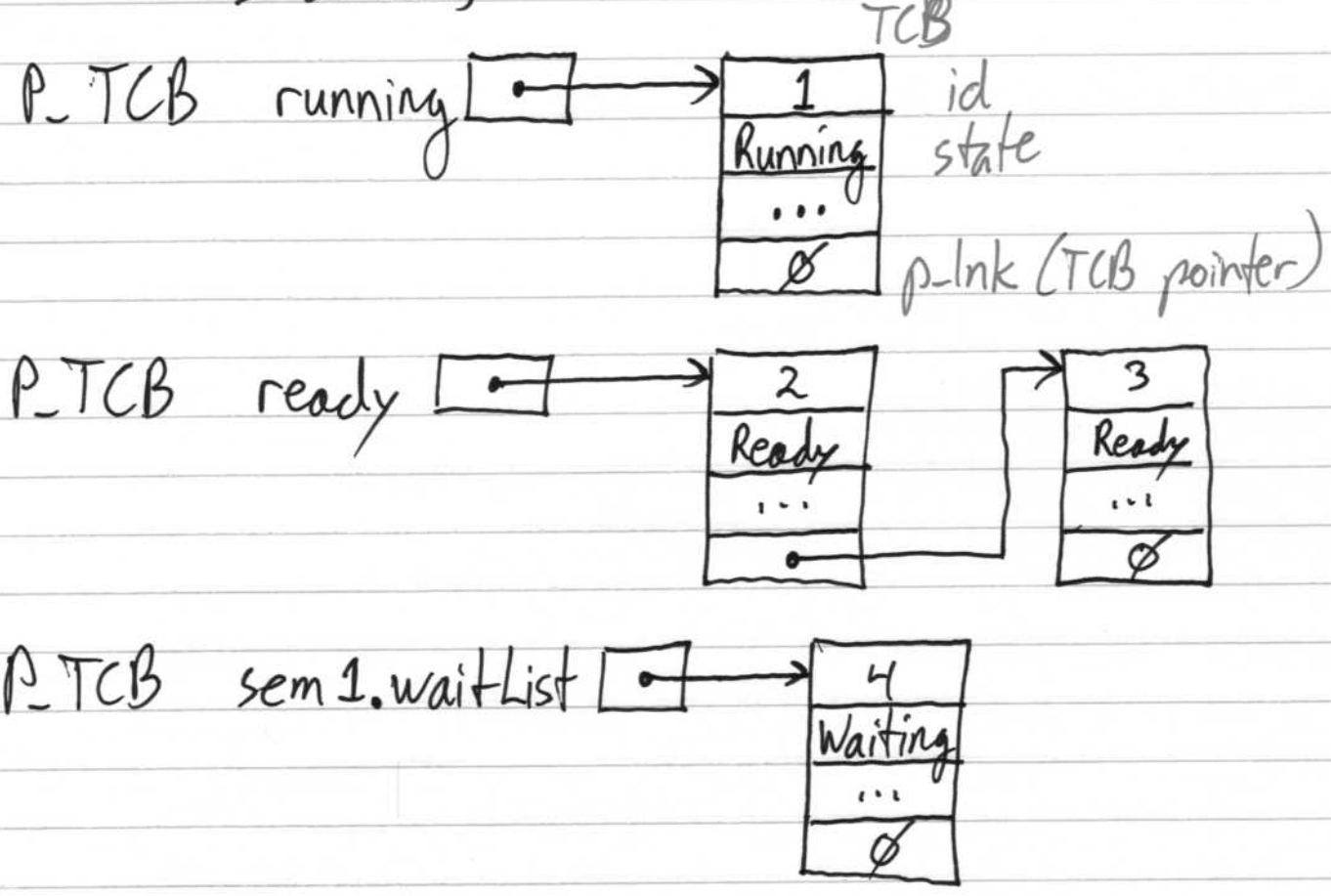


- a better implementation is to block the waiting task (allowing another task to run), and signal() selects one block task (if any) to unblock



- each semaphore now has a wait list associated with it

e.g.
```
typedef struct {
    int32_t s;
    P_TCB waitList;   task control block pointer
} sem_t;
```



- suppose that Task 1 waits on sem1 which is not available ($s \leq 0$)
  - Task 1 is moved to the tail of sem1.waitList
  - Task 1's state changes to Waiting
  - Task 2 is moved from the head of the ready list to the running "list"
  - Task 2's state changes to Running

7) CMSIS-RTX Semaphores

- two types: general semaphore (counting semaphore)
  mutual exclusion semaphore (mutex)

- general semaphore use

```
#include <cmsis_os.h>

osSemaphoreId  sem;
osSemaphoreDef(sem);
sem = osSemaphoreCreate(osSemaphore(sem), 0);
                                          initial count
    - returns NULL on error
osSemaphoreRelease(sem);                  "signal"
    - returns osStatus (osOK or error code)
osSemaphoreWait(sem, osWaitForever);
                     timeout in ms (0=no wait)
    - returns # of tokens (0 = no token, -1 = error)
    - cannot be called by exception handlers (ISRs)
```

- mutex use
  - semaphore count is 1 or 0

```
osMutexId lock;
osMutexDef(lock);
lock = osMutexCreate(osMutex(lock));   initial count = 1
```
  - returns NULL on error

```
osMutexWait(lock, osWaitForever);   "acquire"
osMutexRelease(lock);
```

neither can be called by an exception handler

- RTX mutexes are recursive
  - the same task can acquire the mutex multiple times without blocking
  - it must release the mutex an equal number of times
  - it is a convenience feature (not necessary)
  - the problem is that it can encourage holding the mutex too long
  - you should hold a mutex for as little time as possible to maximize concurrency

c.f. David Butenhof re: recursive mutex

- RTX mutexes check if the task owns the mutex (i.e. has acquired it) on release

- CMSIS-RTOS v1 - no priority inheritance
- CMSIS-RTOS v2 - priority inheritance

8) Priority Inversion

- priority inversion occurs when a higher priority task is indirectly blocked from running by a lower priority task
- it requires 3+ tasks to occur

e.g. Mars Pathfinder Reset Problem [10.5.1]
- used VxWorks RTOS
  - 3 tasks:
    1. bus