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

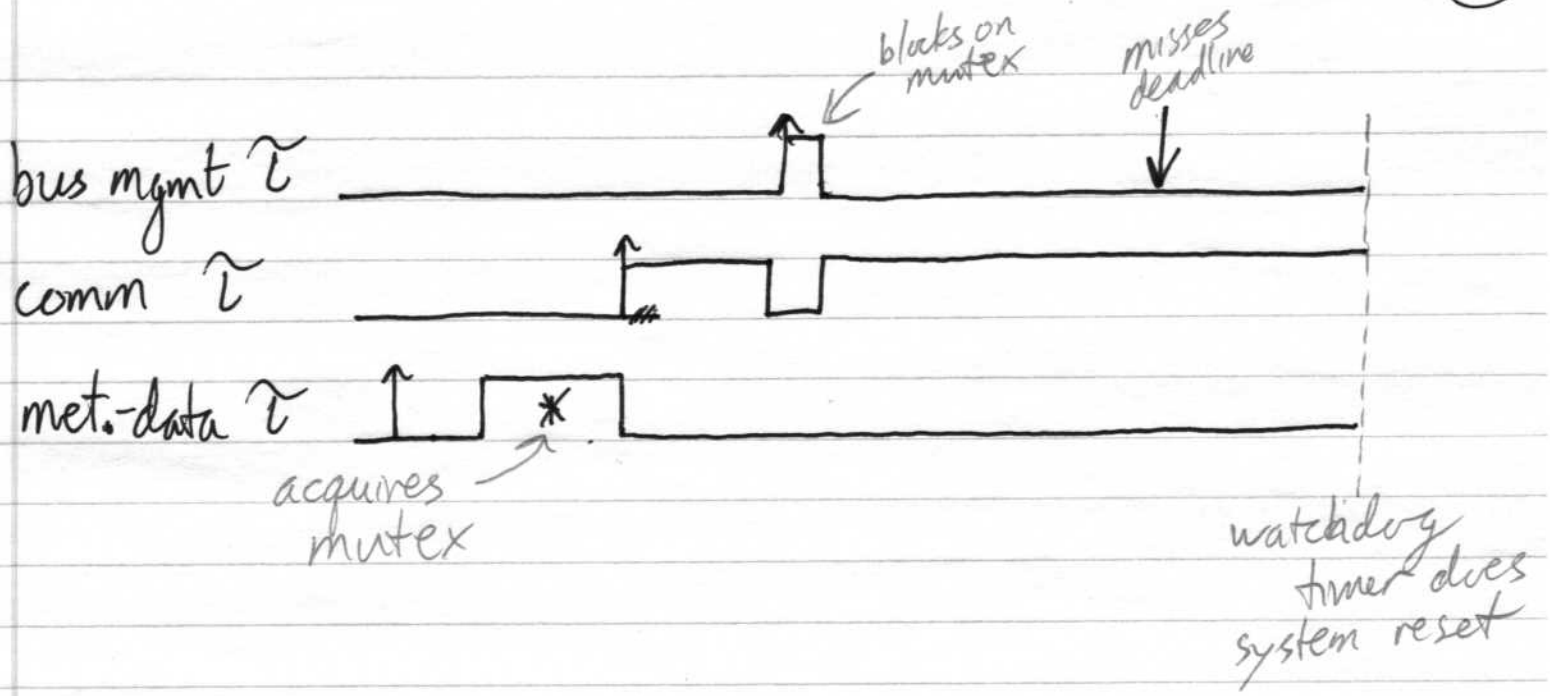
e.g. Mars Pathfinder Reset Problem [10.5.1]
- used VxWorks RTOS

- 3 tasks:

① bus management task
- high priority, executes frequently
- acquires mutex before accessing the bus

② communication task
- medium priority, executes infrequently
- long execution time

③ meterological data gathering task
- low priority, executes infrequently
- also acquires mutex before accessing bus



- with priority inheritance, the priority of the task holding the mutex is temporarily promoted to that of the highest priority task blocked on the mutex

9) Deadlock [11.1]

- deadlock occurs when there is a cycle of tasks holding resources that are needed by other tasks and waiting on a resource held by another task (all tasks are blocked)

- access to shared resources such as timers, data, I/O devices is typically protected by mutexes

- the 4 conditions required for deadlock:

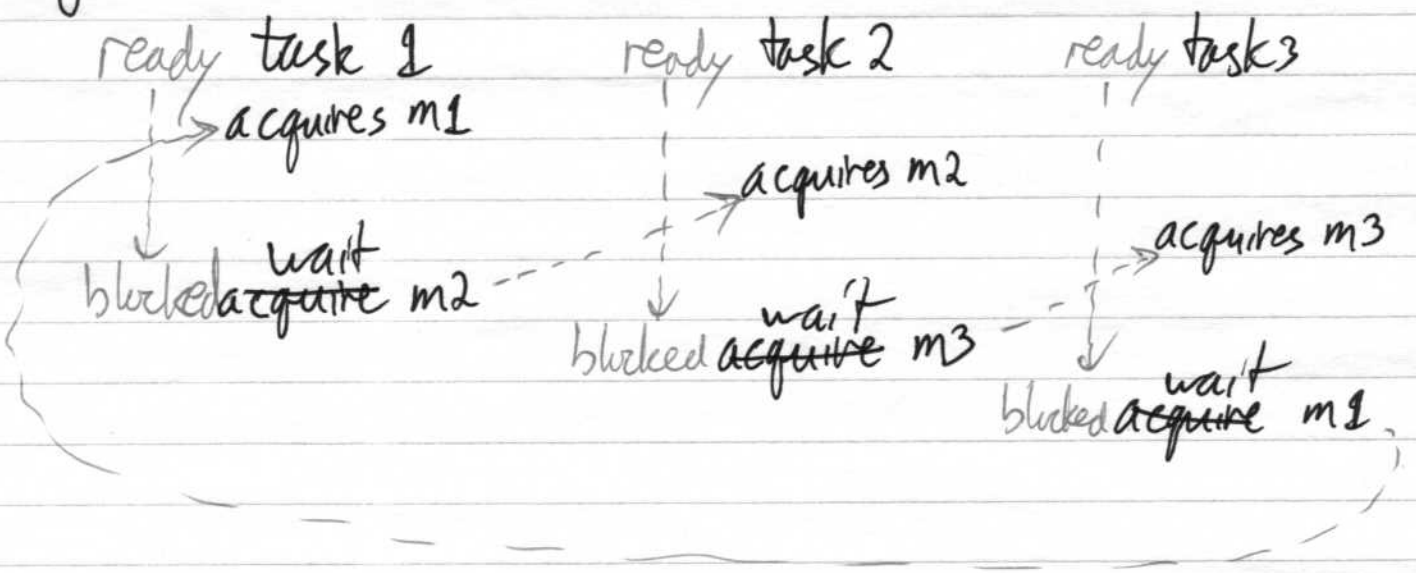
  (1) mutual exclusion - mutexes guard access to shared resources

  (2) hold and wait - task holds 1+ mutexes and waits on another

  (3) no preemption of resources - an acquired mutex can only be released by its owner

  (4) circular wait - see below

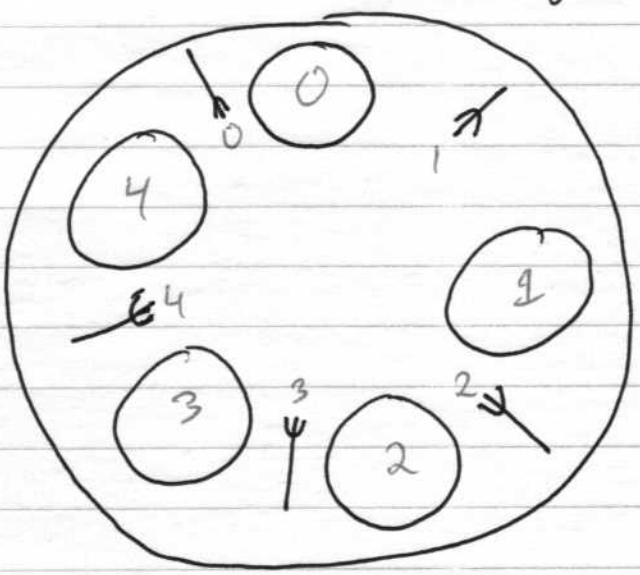
eg. 3 tasks, 3 mutexes m1, m2, m3



- to prevent deadlock ~~deadlock~~ we need to eliminate at least one of the 4 conditions

9.1) Dining Philosophers Problem

- created by Edsger Dijkstra for a grad course exam



- 5 philosophers sit in front of 5 bowls of spaghetti
- between each bowl is 1 fork
- philosophers alternately think and eat
- they need both the fork to the left and the fork to the right to eat
- design an algorithm so no philosopher starves

- model the philosophers as tasks
- model the forks with mutexes

e.g. osMutexId fork[5];

- the main function:
  - initialize the mutexes
  - create 5 philosopher tasks ~ osThreadCreate(osThread(philosopher), (void*) id);

```
void philosopher (void *id_arg) {
    uint32_t id = (uint32_t) id_arg;   id ∈ [0,4]
    while(true) {
        // think for a while
        osDelay( rand() % 5000 + 1);   // in stdlib, 1-5000 ms

        ⇒ /*
            pick up forks */

        // eat for a while
        osDelay (rand() % 5000 + 1);

        ⇒ /*
            put down forks */
    }
}
```

- first attempt:

pick up forks

```
osMutexWait(fork[id], osWaitForever);
osMutexWait(fork[(id+1)%5], osWaitForever);
```

put down forks

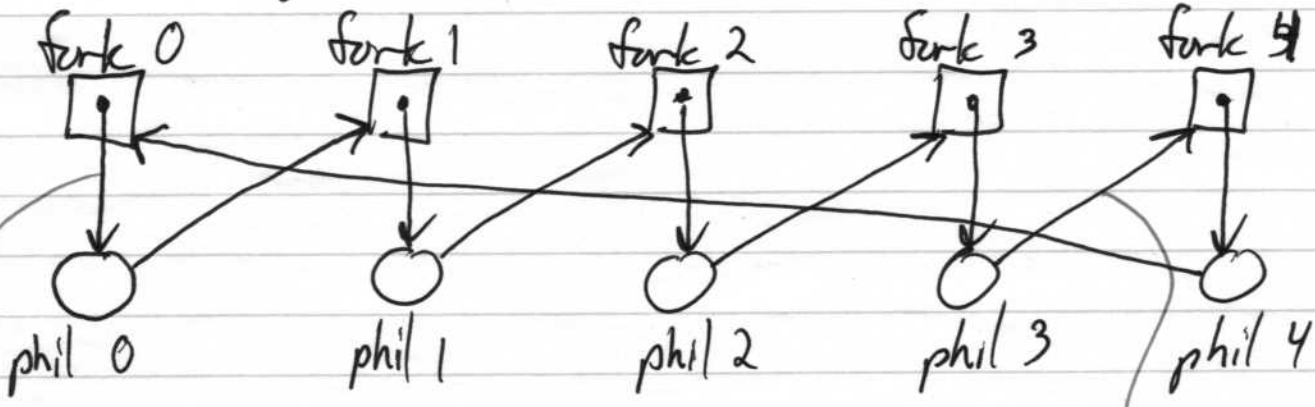
```
osMutexRelease(fork[id]);
osMutexRelease(fork[(id+1)%5]);
```

- can deadlock occur? yes

phil 0	phil 1	phil 2	phil 3	phil 4
acquire fork 0	acquire fork 1	acquire fork 2	acquire fork 3	acquire fork 4
wait fork 1	wait fork 2	wait fork 3	wait fork 4	wait fork 0

circular wait

- ~~alternate diagrams~~ resource allocation graph



- arc from resource to task means it is acquired
- arc from task to resource means it is waiting

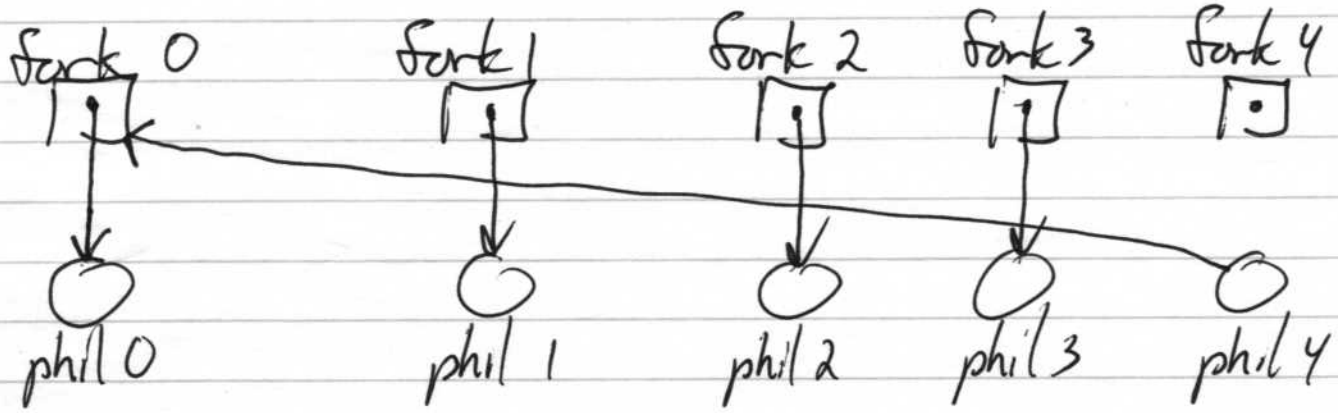
- to avoid/break deadlock we need to eliminate 1 of the 4 conditions

e.g. (4) circular wait - can avoid by using the Ordered Resource Policy

Ordered Resource Policy (ORP): assign an order to the resources, and all tasks acquire their resources following that order
- e.g. pick up lower-numbered fork first

pickup forks
```
uint32_t first = id==4 ? 0 : id;
uint32_t second = id==4 ? 4 : id+1;
osMutexWait(fork[first], osWaitForever);
osMutexWait(fork[second], osWaitForever);
```

put down forks
```
osMutexRelease(fork[first]);
osMutexRelease(fork[second]);
```



- no deadlock; no starvation

10) Starvation

- a thread is unable to access a shared resource indefinitely
  - happens when greedy threads monopolize the resource

- starvation could happen if we solved dining philosophers by picking up both forks at once

e.g.

```
sem_t mutex;
uint8_t fork[5] = {0};

pick up forks
bool haveForks = false;
while(!haveForks) {          <- wait(&mutex);
    if(fork[first]==0 && fork[second]==0){
        fork[first]=1;
        fork[second]=1;      } fork[first]=fork[second]=0;
        haveForks = true;
    }
}                            <- signal(&mutex);
```

t	phil 0	phil 1	phil 2
	pick up forks(0,1)		
		try to pick up	
	put down forks		
			pick up forks(2,3)
		try to pick up	
			put down forks
	pick up forks		
		try to pick up	
		⋮	

- no deadlock; phil 1 starves

10.1) Readers-Writers Problem

- there are multiple readers and writers of shared data
  e.g. multiple tasks sharing a file, database, shared data
  - requirements:
    1. concurrent reader access
    2. exclusive writer access
    3. no starvation

- solution
  - consider that the shared data is in a room

```
sem_t roomEmpty;
init(&roomEmpty, 1);
```

writer task

```
wait(&roomEmpty);
... perform writing ...
signal(&roomEmpty);
```

reader task

```
wait(&roomEmpty);
... perform reading ...
signal(&roomEmpty);
```

- which requirements are met?

concurrent reader access ✗
exclusive writer access ✓
no starvation ✓

- next solution [Little Book of Semaphores]
  - if a reader is in the room, allow other readers in

```
sem_t roomEmpty;
sem_t mutex;
uint32_t readers; // tracks # readers in the room

init(&roomEmpty, 1);
init(&mutex, 1);
readers = 0;
```

writer task

```
wait(&roomEmpty);
... perform writing ...
signal(&roomEmpty);
```

reader task

```
wait(&mutex);
    readers++;
    if(readers == 1)
        wait(&roomEmpty);
signal(&mutex);

... perform reading

wait(&mutex);
    readers--;
    if(readers == 0)
        signal(&roomEmpty);
signal(&mutex);
```

- requirements met:
  - concurrent readers ✓
  - exclusive writers ✓
  - no starvation ✗

- starvation can occur if readers keep arriving before the room empties resulting in writers waiting forever
- solution: add a turnstile (see barrier) to prevent new readers from barging ahead of waiting writers
  - see A3

Communication

① Producer/Consumer Problem

- common pattern in embedded systems

producers	consumers
menu task (gui)	LCD draw task
system health monitor	LCD draw task
ADC isr	signal transform task
wheel rotation sensor (sensor ISR)	ABS controller

② Handshake Protocol

- 1 message buffer

```
const uint32_t bufSize = 512;
uint8_t buf[bufSize];
sem_t empty, full;

init(&empty, 1);
init(&full, 0);
```

producer (task)

```
while(true) {
    wait(&empty);
    ... write to buf
    signal(&full);
}
```

consumer task

```
while(true) {
    wait(&full);
    ... read from buf
    signal(&empty);
}
```

- disadvantage: producer and consumer cannot overlap