Inter-task Communication

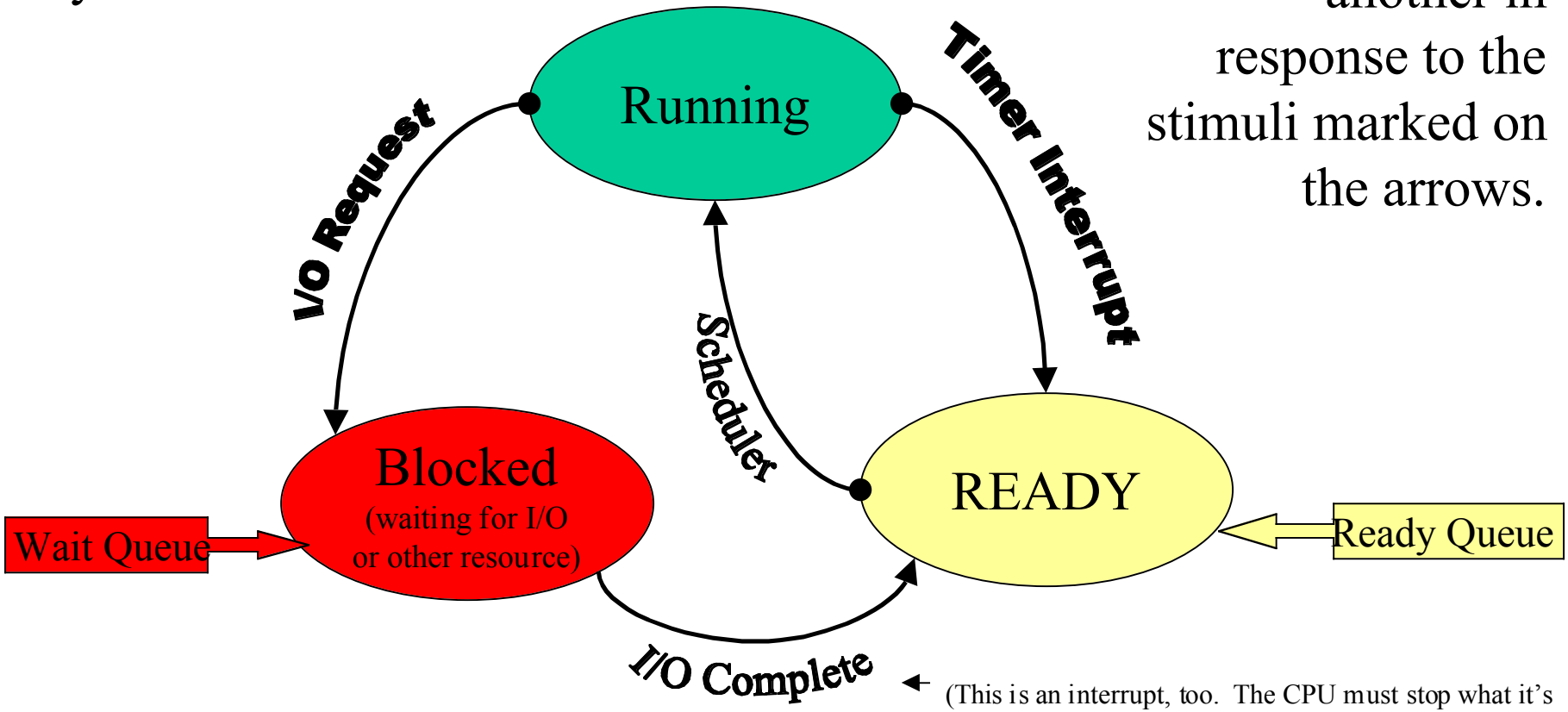
04/27/01 Lecture # 29 16.070

- Task state diagram (single processor)
- Intertask Communication
  - Global variables
  - Buffering data
  - Critical regions
- Synchronization
  - Semaphores
  - Mailboxes and Queues
  - Deadlock
- Readings: Chapter 7 in Laplante

Task state diagram

A process goes through several states during its life in a multitasking system.

Tasks are moved from one state to another in response to the stimuli marked on the arrows.



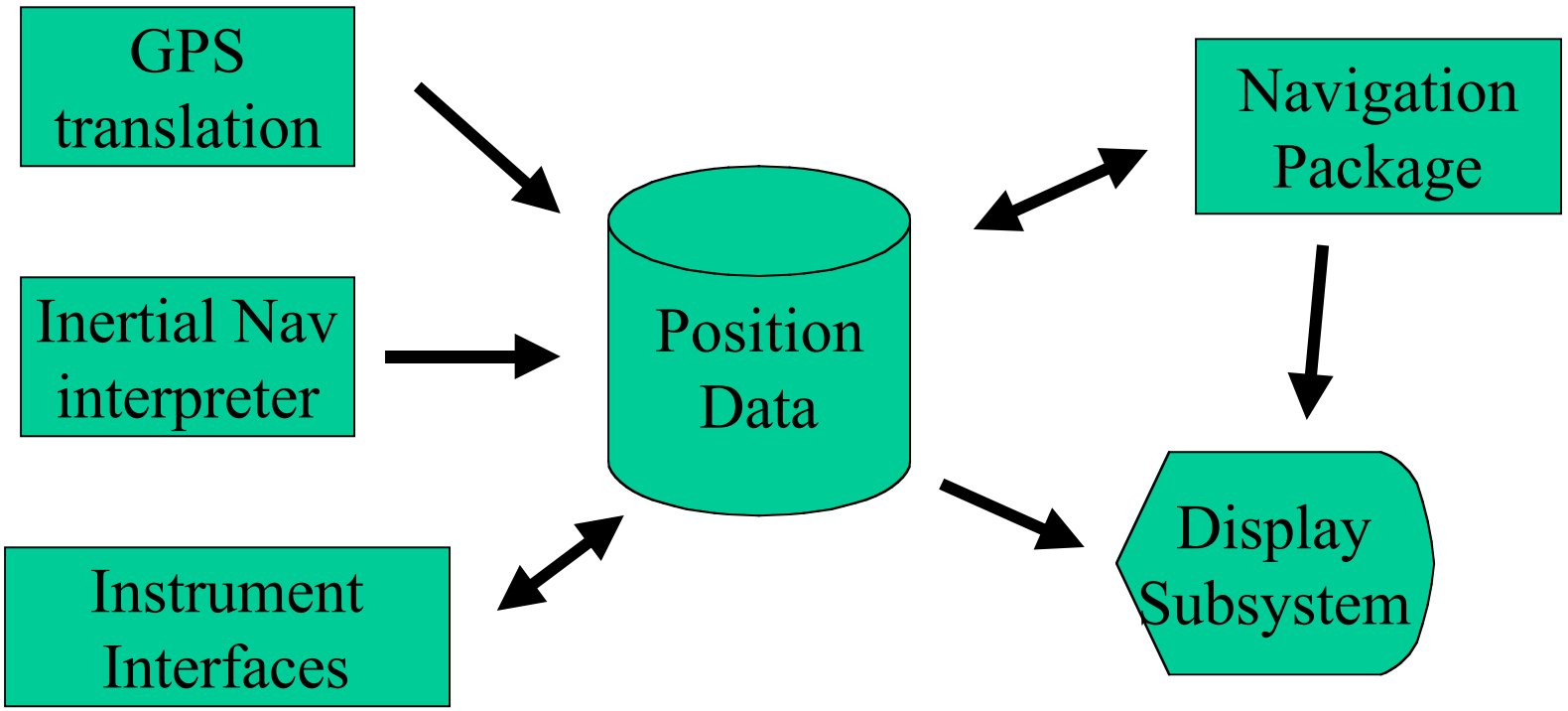
State Diagram description

- Any tasks that are ready to run sit on the *ready queue.* This queue may be prioritized so the most important task runs next.

- When the scheduler decides the current task has had enough time on the CPU, either because it finished or its time slice is up, the “Running” task is moved to the “Ready” queue. Then the first task on the “Ready” queue is selected for “Running”.

- If the “Running” task needs I/O or needs a resource that is currently unavailable, it is put on the “Blocked” queue. When its resource becomes available, it goes back to “Ready”.

Tasks don't work in isolation from each other. They often need to share data or modify it in series



Inter-task communication examples

- Since only one task can be running at one time (remember the book analogy), there must be mechanisms for tasks to communicate with one another
  - A task is reading data from a sensor at 15 hz. It stores 1024 bytes of data and then needs to signal a processing task to take and process the data so it has room to write more.
  - A task is determining the state of a system- i.e. Normal Mode, Urgent Mode, Sleeping, Disabled. It needs to inform all other tasks in the system of a change in status.
  - A user is communicating to another user across a network. The network receive task has to deliver messages to the terminal program, and the terminal program has to deliver messages to the network transmit task.

Inter-task Communication

- Regular operating systems have many options for passing messages between processes, but most involve significant *overhead* and aren't deterministic.
  - *Pipes, message queues, semaphores, Remote Procedure Calls, Sockets, Datagrams, etc.*
- In a RTOS, tasks generally have direct access to a common memory space, and the fastest way to share data is by sharing memory.
  - In ordinary OS's, tasks are usually prevented from accessing another task's memory, and for good reason.

Global Variables:

an example in pseudocode

```
int finished = 0;

main()
{
   spawn( task1 );
   spawn( task2 );
   spawn( task3 );

   while( finished !=3)
   {
    ;
   }
   printf(" done ");
}
```

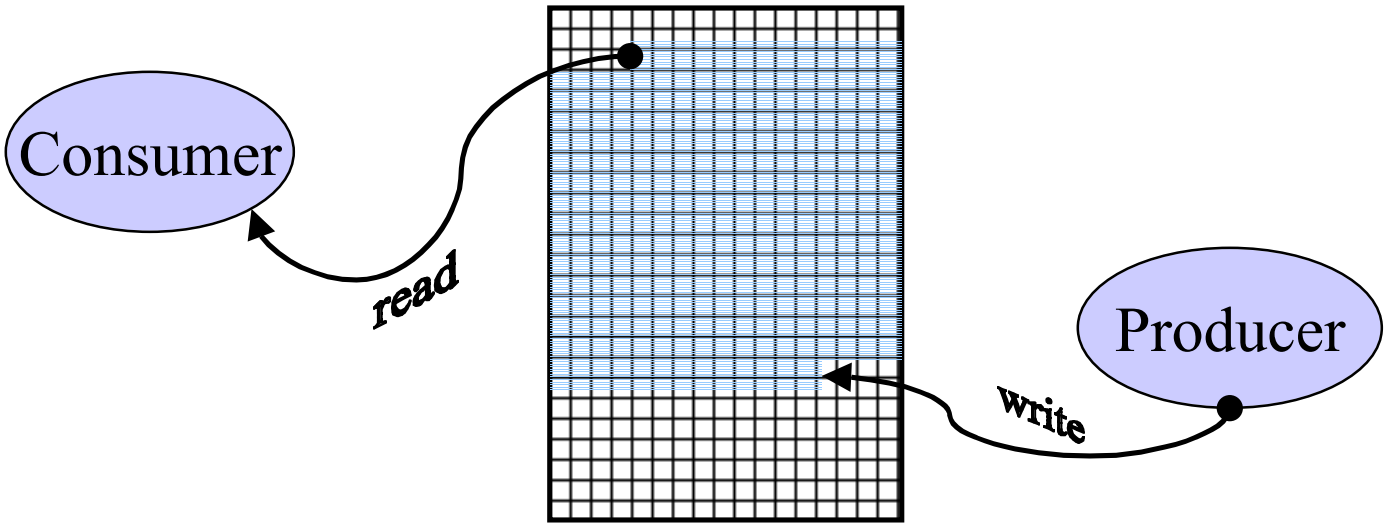
```
void task1 (void)
{
   compute_pi_to_a zillion_places();
   finished++;
}

void task2 (void)
{
   solve_world_hunger();
   finished++;
}

void task3 (void)
{
find_out_why_white_shirts_give_you_
black_belly_button_lint();
   finished++;
}
```

Mailboxes

- Post() - write operation- puts data in mailbox
- Pend() - read operation- gets data from mailbox
- Just like using a buffer or shared memory, except:
  - If no data is available, pend() task is suspended
  - Mutual exclusion built in:
    if somebody is posting, pend() has to wait.
- No processor time is wasted on polling the mailbox, to see if anything is there yet.
- Pend might have a timeout, just in case

Buffering Data

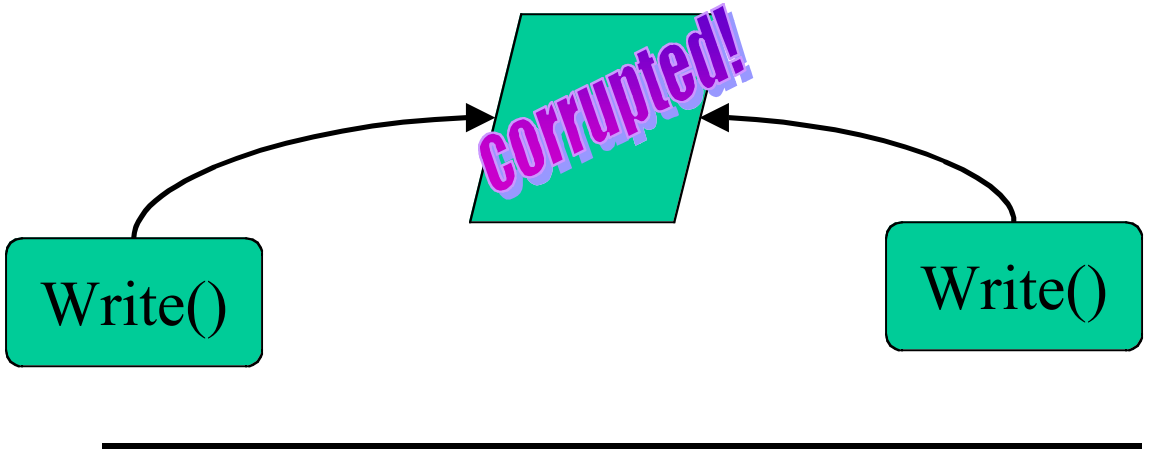
- If you have a producer and a consumer that work at different rates, a buffer can keep things running smoothly
  - As long as buffer isn't full, producer can write
  - As long as buffer isn't empty, consumer can read



Shared Memory and Data Corruption

•Shared memory can be as simple as a global variable in a C program, or an OS-supplied block of common memory.

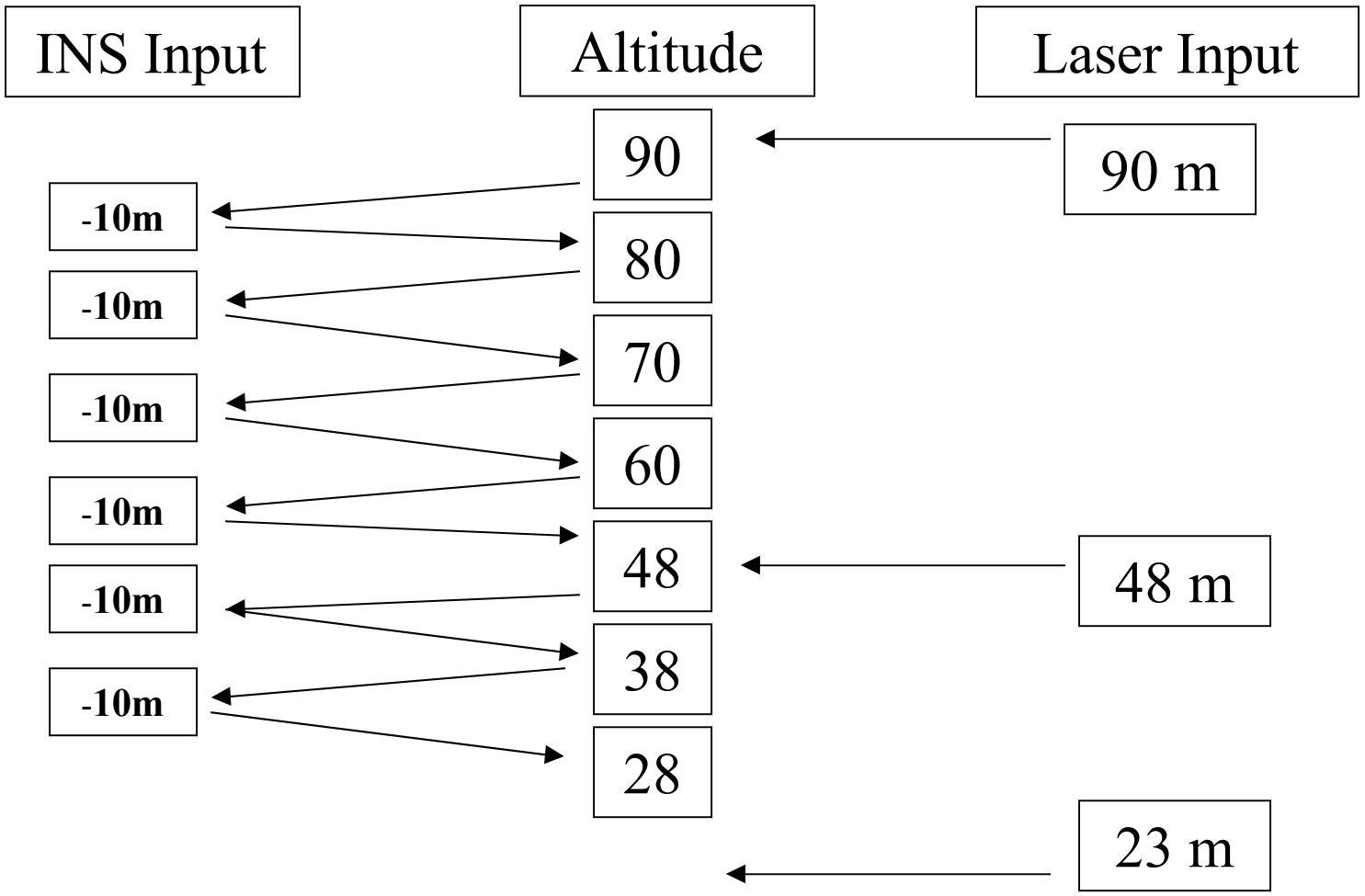
•In a single-task program, you know only one function will try to access the variable at a time.



Let's look at a navigation system:

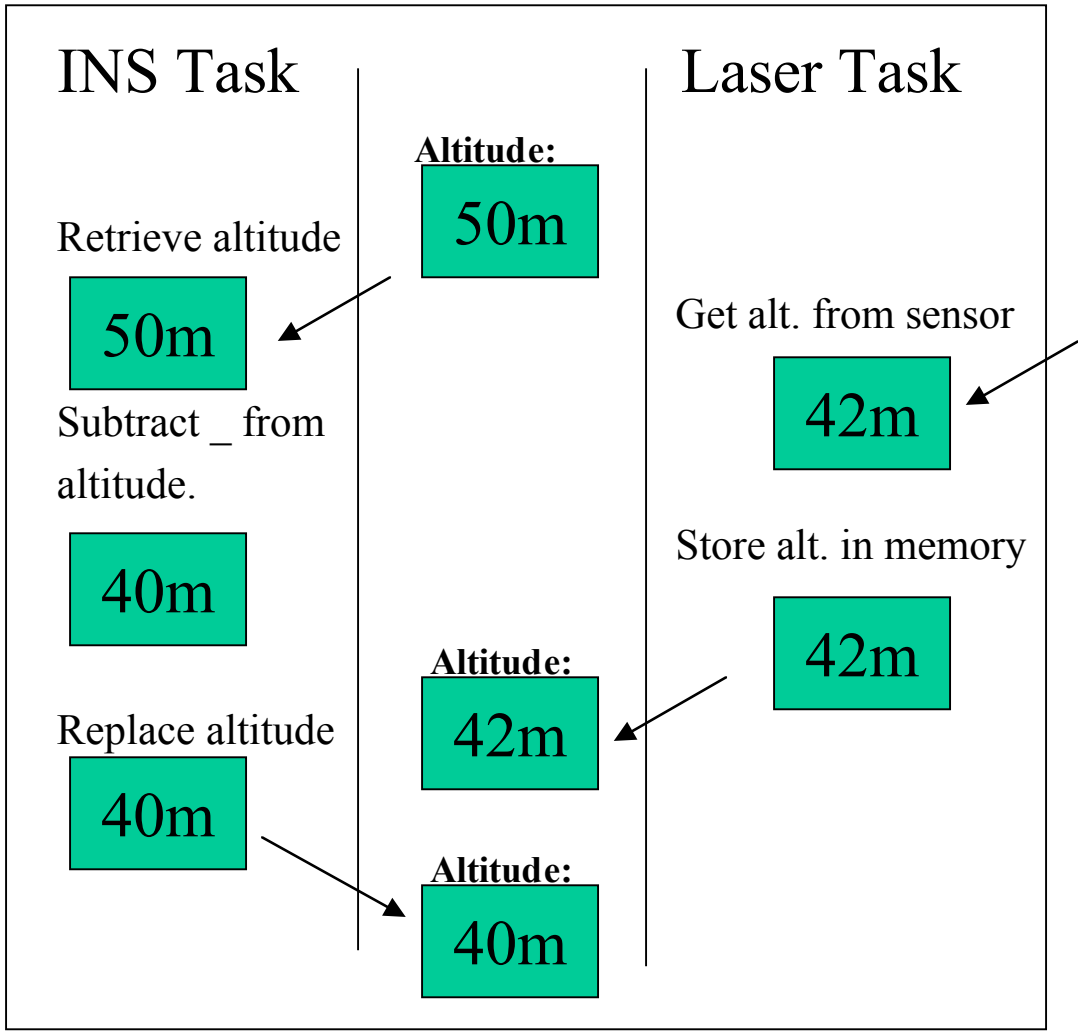
- We use a laser-based rangefinder to get altitude readings as available (approx. once every 5 seconds).
- We add a redundant system, an inertial navigation system, to update the altitude once a second:

2 tasks sharing the same data

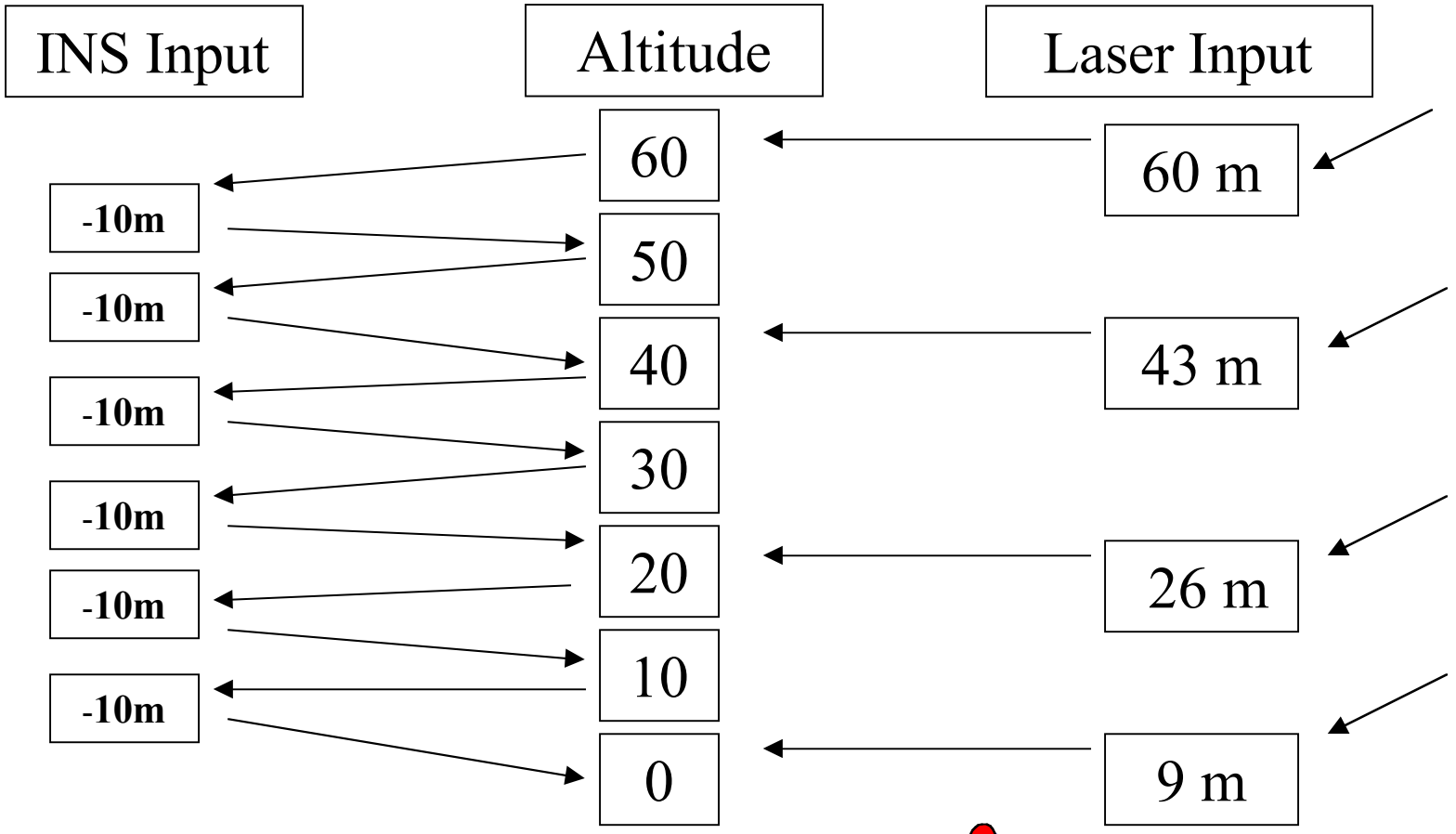


Shared memory conflict:

- The INS executes several instructions while updating the altitude:
  - Get stored altitude
  - Subtract _ altitude
  - Replace altitude with result
- One task may interrupt another at an arbitrary (possibly Bad™) point.



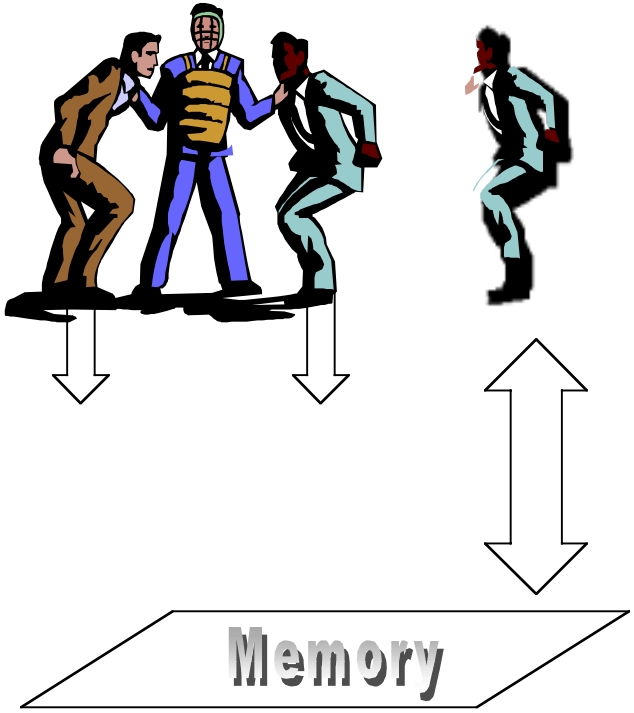
Timing Problem:



Avoiding Conflict

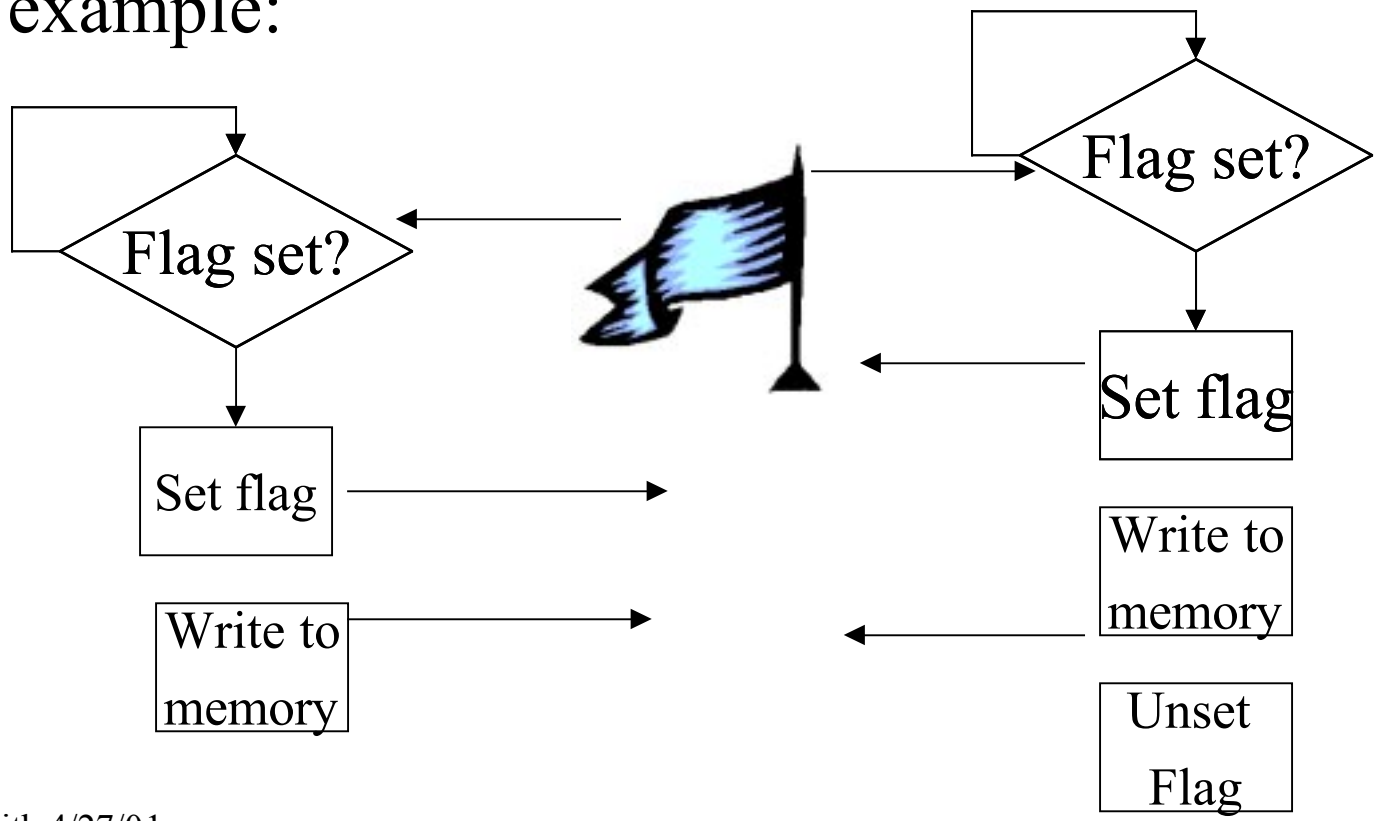
We need to be careful in multi-tasked systems, especially when modifying shared data.

We want to make sure that in certain *critical sections* of the code, no two processes have access to data at the same time.



Mutual Exclusion

- If we set a flag (memory is busy, please hold), we can run into the same problem as the previous example:



Atomic Operations

- An operating system that supports multiple tasks will also support atomic *semaphores.*
- The names of the functions that implement semaphores vary from system to system (

test-set/release; **lock/unlock**; **wait/signal**; **P**()/**V**())

- The idea: You check a "lock" before entering a critical section. If it's set, you wait. If it isn't, you go through the lock and unset it on your way out.
- The word *atomic* means checking the lock and setting it only takes one operation- it can't be interrupted.

Semaphore Example

```
TelescopeImageUpdate()
{
  if( time_is_now() )
  {
  lock( image_map );
  …
  update( curr_image[] );
  unlock( image_map );
  …
}
```

```
ImageTransmit()
{
  …
  if( command == XMIT )
  {
  lock( transmitter );
  lock( image_map );
  …
  broadcast( curr_image[] );
  unlock( image_map );
  unlock( transmitter );
  }
  …
}
```

Deadlock!

```
ImageTransmit()
{
  …
  if( command == XMIT )
  {
  lock( transmitter );
  lock( image_map );
  …
  broadcast( curr_image[] );
  unlock( image_map );
  unlock( transmitter );
  }
  …
}
```

Waiting on image_map

```
Process_Image_Weights()
{
  …
  lock( image_map );
  …
  color_map( curr_image[] );
  lock( transmitter );
  …
  broadcast(colored_image[]);
  unlock( transmitter );
  unlock( image_map );
  …
}
```

Waiting on transmitter

Deadlock: detection and avoidance

- Cannot always be found in testing
- Four conditions necessary
  - Area of mutual exclusion
  - Circular wait
  - Hold and wait
  - No preemption
- Some well-known solutions exist
  - Make all resources sharable
  - Impose ordering on resources, and enforce it
  - Force a task to get all of its resources at the same time or wait on all of them
  - Allow priority preemption

DRAPER LABORATORY

Other ways around synchronization problems

- Avoidance: Only write single-task programs or programs that don't use shared memory
- Ostrich method: Ignore the problem completely, assuming it won't happen often, or at least not often enough for your customers to sue you
- Brute force: Disable interrupts completely during "critical section" operations

Summary

- Buffering data can smooth out the interaction of a producer that generates data at one rate and a consumer that eats at another.
- Intertask communication can be tricky- if your operating system supports high-level communication protocols, and they are appropriate for your task, use them!
- If you use a flag to indicate a resource is being used, understand why checking and setting the flag needs to be atomic.

- For Next time: Read sections 11.1, 11.2 (intro section only) 11.3, 11.4