Last Time

- Communication through buses
  - System buses
  - Backplane buses
  - I/O buses
- I^2C Bus
  - 2-wire
  - Multi-master
- Project 4

Today

- Operating systems
- Multitasking

Administrivia

- Lab 4: due next Thursday @5:00
  - Demo
  - Group report
  - Personal report
- Lab hours today: 10:30-3:30
- Homework 6 out next Tuesday

Operating Systems

Operating systems are all about abstraction…

- Typically multiple layers in the abstraction
- Each layer hides some of the details of what is below
- A layer will often unify several components at lower layers

Common OS Layers

- Device drivers: direct interface to hardware
- Kernel: system management activities
- Middleware: provides general high-level services that may be used by custom applications

The precise divisions will vary depending on the OS and system

Device Drivers

- Small pieces of code that interface to specific types of hardware
- Typically designed to execute very quickly
- Provide a more uniform interface to the hardware
  - Different implementations “look” the same to the software above

Device Drivers

Example

- Different disk manufacturers use different protocols for reading/writing from/to their disks
- A device driver for one type of disk can make it appear to the rest of the system as any other disk

Kernel

System management services:

- Process management: allows us to separate our programs into multiple, separate tasks (or processes)
- Memory management: controlled sharing of memory between processes
- I/O management:
  - Device sharing (across processes)
  - Providing key abstractions (e.g., files)

Processors to Processes

Processor: the hardware that executes a program

- A single processor essentially does only one thing at any one time

Process: the computational abstraction

- Consists of: program, memory, stack, program counter, etc.
- But:
  - It is a passive entity
  - Many can exist at any one time

Handling Multiple Tasks/Processes

With an interrupt, we implemented two separate tasks:

- Main code body (our “something else”)
- Interrupt handler: deal with the external or internal event

Handling Multiple Tasks/Processes

With an interrupt, we implemented two separate tasks:

- Main code body (our “something else”)
- Interrupt handler: deal with the external or internal event

- These are essentially separate entities
- But: with a little bit of communication between them

Handling Multiple Tasks

With a complex system:

- Often have many different tasks to be performed
- These tasks can have different timing requirements:
  - How often they must be performed
  - How quickly they must respond to an event

The Multi-Tasking Abstraction

This abstraction is key to building complex systems

- We can construct our system as a set of compartmentalized modules
- Each module can be implemented and tested separately
- It is easy to “mix and match” modules depending on the application

The Multi-Tasking Abstraction

This abstraction is key to building complex systems

- Each process has the “illusion” of owning the processor all of the time
- Allows for efficient use of the CPU and other system resources

Multi-tasking

- At any one time, a single process is in control of (or “owns”) the processor
  - We refer to this process as being in a **running state**
- All other processes are either:
  - In a **waiting state**: waiting on some external or internal event
  - In a **ready state**: ready to execute when the processor is free

An Example: USC AFV

Sensors:

- Downward-oriented sonars: height and attitude
- Compass: yaw direction
- Rotor encoder: rotational velocity
- Downward-looking camera: position on field

An Example: USC AFV

Actuators:

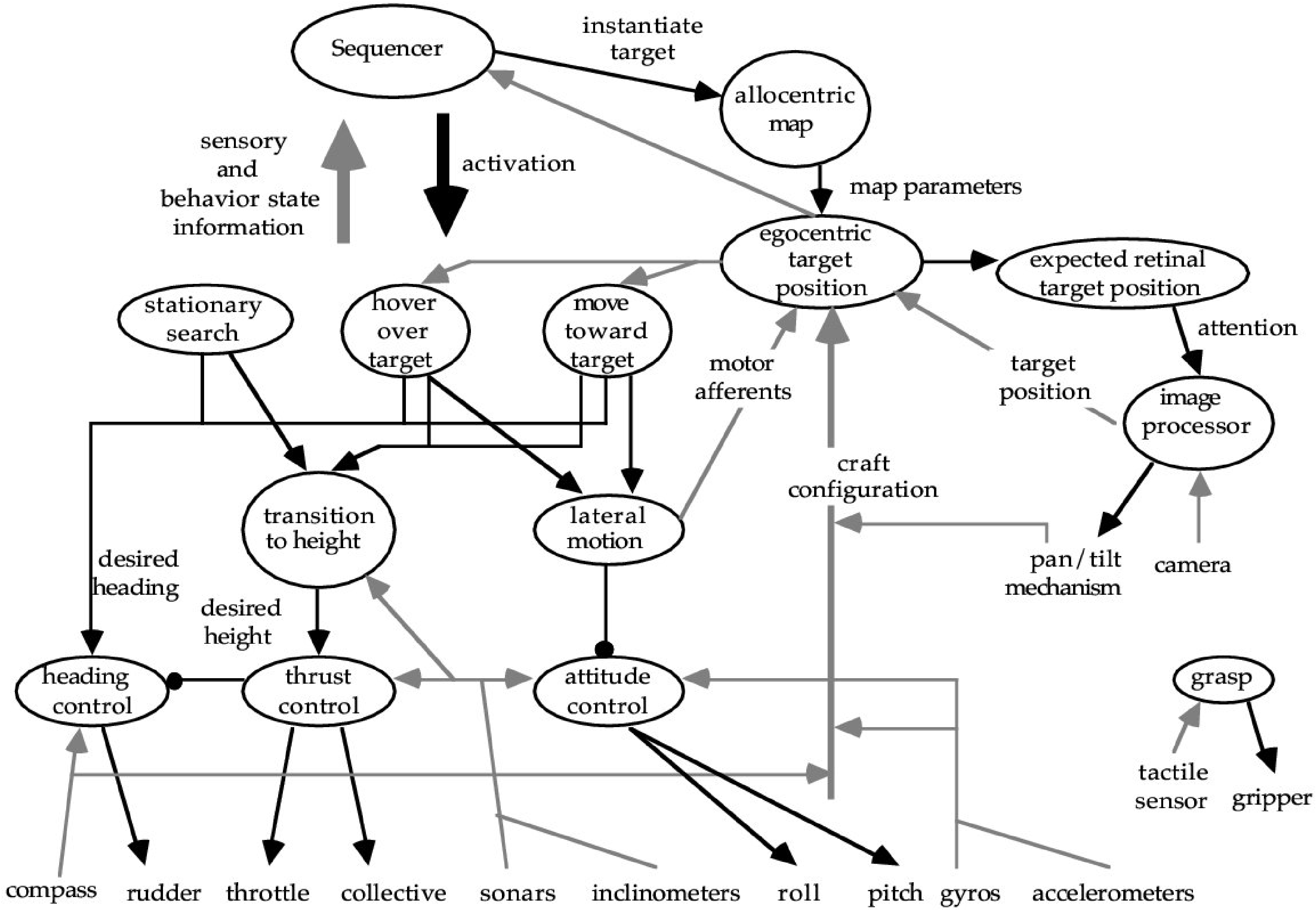
- Rotor collective
- Rotor torque
- Rotor pitch
- Rotor yaw
- Rudder

An Example: USC AFV

Tasks include:

- Thrust control
- Attitude control
- Heading control
- Move to height
- Search for target
- Hover over target
- Planner

AFV Process Architecture



Multi-Tasking Components

- Process control block (PCB): data structure that describes the process
- Scheduling: deciding which process to execute now
- Inter-task communication: moving data between processes
- Synchronization: mechanism for safely coordinating the actions of two or more processes

Process (or Task) Control Block

- Process identifier (PID)
- Process state (running, waiting, ready)
- Priority
- Information about memory allocated to the process (including primary memory and the stack)
- Register state (including program counter)

Operations on a Process

- Creation (fork/exec/spawn)
- Suspend: stop a process temporarily
- Resume: undo the suspend
- Destroy: stop executing the process and deallocate its memory

Last Time

- Operating Systems
  - Device drivers
  - Kernel
  - Middleware
- Processes

Today

- More on processes
  - Blocking
  - Context switching
- Scheduling policies
  - How to select the next process to execute?
  - Queues
  - Priorities
  - Preemption

Administrivia

- Project 4 due Thursday
  - Demonstration
  - Group report
  - Personal report
- Homework 6 out tonight
- Next Tuesday: virtual guest visit

Jim Montgomery
Robotic Software Systems Group
NASA/Jet Propulsion Laboratory

  - Readings will be posted on D2L

Multi-tasking

- At any one time, a single process is in control of (or “owns”) the processor
  - We refer to this process as being in a **running state**
- All other processes are either:
  - In a **waiting state**: waiting on some external or internal event
  - In a **ready state**: ready to execute when the processor is free

Causing a Process to Block

A variety of situations can cause a process to move into a **blocked** or **waiting** state:

- Waiting for an I/O operation to complete
- Waiting for a timer to expire (e.g., if the process is to execute once every 10 ms)
- Waiting for another process to provide information or to complete its current operation

Switching Between Processes

The OS will switch from one process to another at some instant in time. This can happen for a variety of reasons (and depending on the OS)

- The current process blocks on some event
- The process no longer needs the processor
- The process has executed for long enough
- A process of higher priority requires the processor

Context Switches

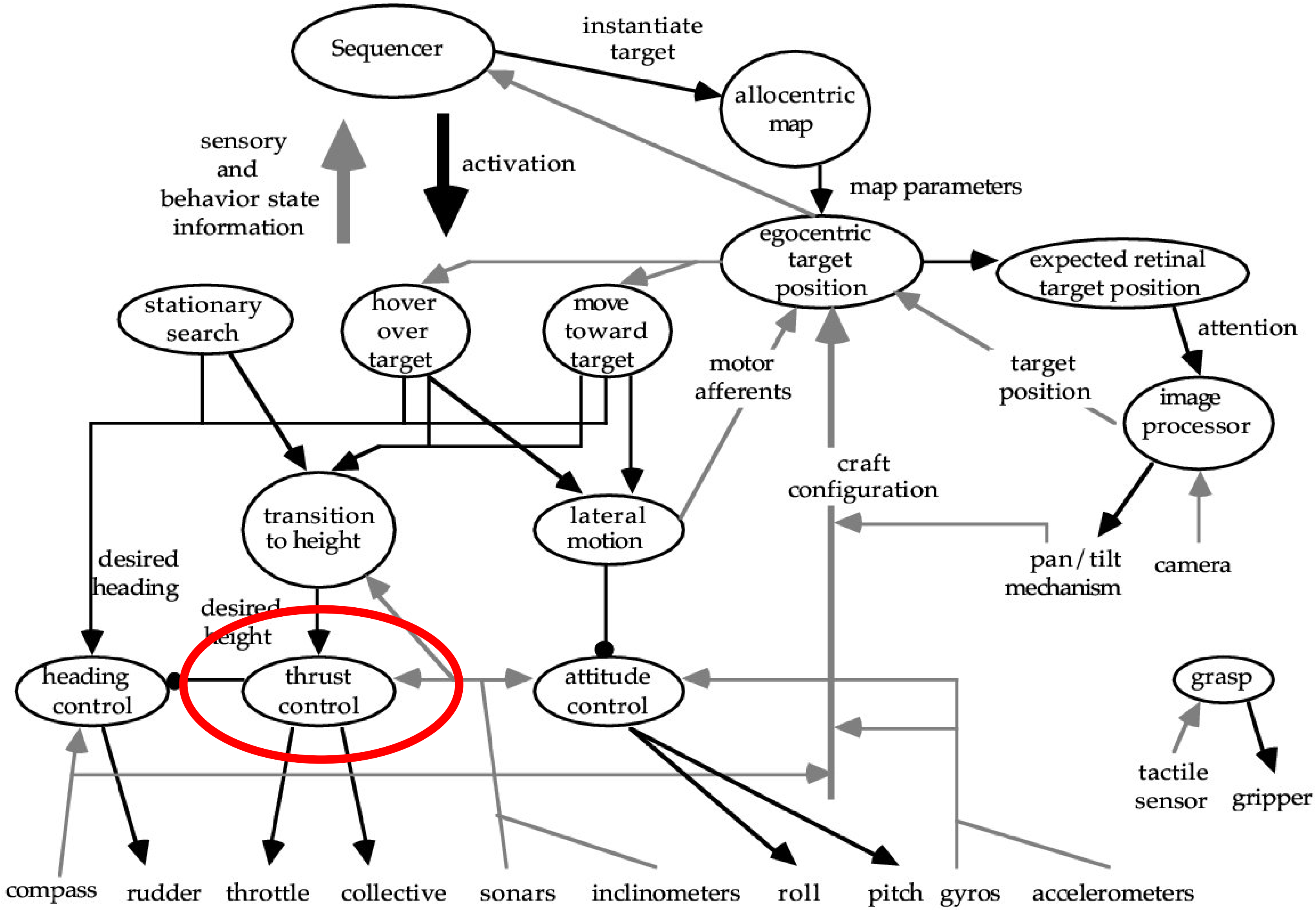
- The state of current process must be saved:
  - Program counter
  - Stack pointer
  - Registers
- The state of the next process is then restored to the processor:
  - Program counter
  - Stack pointer
  - Registers

Processes and Threads

Threads are sometimes called **lightweight processes**

- Memory:
  - Processes have their own, separate memory
  - A set of threads will share memory
- Coordination
  - Processes will often be independent of one-another
  - Threads are typically working together on a common problem

A Process Example



An Example:
Altitude Control Process

“BURTE” kernel

```
void altitude_servo_loop()
{
   set_schedule_interval(10);  // 10ms
   while(1)
        {
              collective = Kp * (height_desired - height)
                                 - Kv * height_velocity;
              set_collective(collective);

              next_interval(); // Wait for the next control
                                    //  cycle
        };
};
```

An Example: Starting the Process

```
main()
{

              :
  pid = create(altitude_servo_loop, 10, 3000);
  start(pid);
              :
};
```

An Example: Starting the Process

```
main()
{
              :
  pid = create(altitude_servo_loop, 10, 3000);
  start(pid);
              :
};
```

Name of function

An Example: Starting the Process

```
main()
{

                :
   pid = create(altitude_servo_loop, 10, 3000);
   start(pid);
                :
};
```

Priority

An Example:
Starting the Process

```
main()
{
               :
   pid = create(altitude_servo_loop, 10, 3000);
   start(pid);
               :
};
```

Size of stack

An Example:
Starting the Process

```
main()
{
                :
   pid = create(altitude_servo_loop, 10, 3000);
   start(pid);
                :
};
```

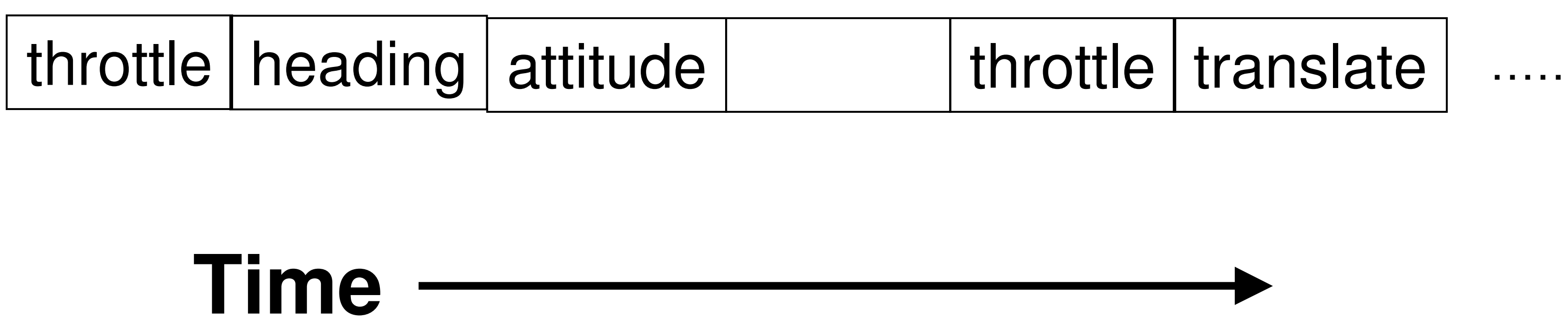
Start the process

Summary

- Process: an piece of code with data and processor state information
- Multi-processing: switching quickly between processes
- Process control block: data structure for process management

Selecting a Process to Execute

Only one process may occupy the processor at any one time…



Selecting a Process to Execute

A **scheduler** is responsible for selecting the next process

- How might we do this?

Scheduling Policies

Only processes in the **ready state** may be selected

- Round robin: rotate between the different processes
- Priority-based: select the highest-priority process that is ready to execute
- Shortest-process-first: select the one that will use the processor for the shortest period of time

- Preemption: interrupt an executing process

Evaluating Scheduling Policies

Metrics for evaluation include:

- **Response time**: time for a process to move from ready to running
- **Turn-around time**: time for a process to move from ready to running and then to leave running
- **Throughput**: number of processes that can be executed in a given period of time
- **Overhead**: the amount of time required by the operating system to perform scheduling

Evaluating Scheduling Policies

Other key concepts:

- **Fairness**: all processes get some access to the processor (and other resources)
- **Starvation**: a process never gets access to the processor (because other processes are occupying it)

Round Robin Scheduling

- Queue: an ordered list of processes that are in a **ready** state
- Selecting the next processes: remove the process from the top of the queue
- Any new processes: add to the end of the queue

Note: the book defines RR as necessarily being preemptive. This is not the case

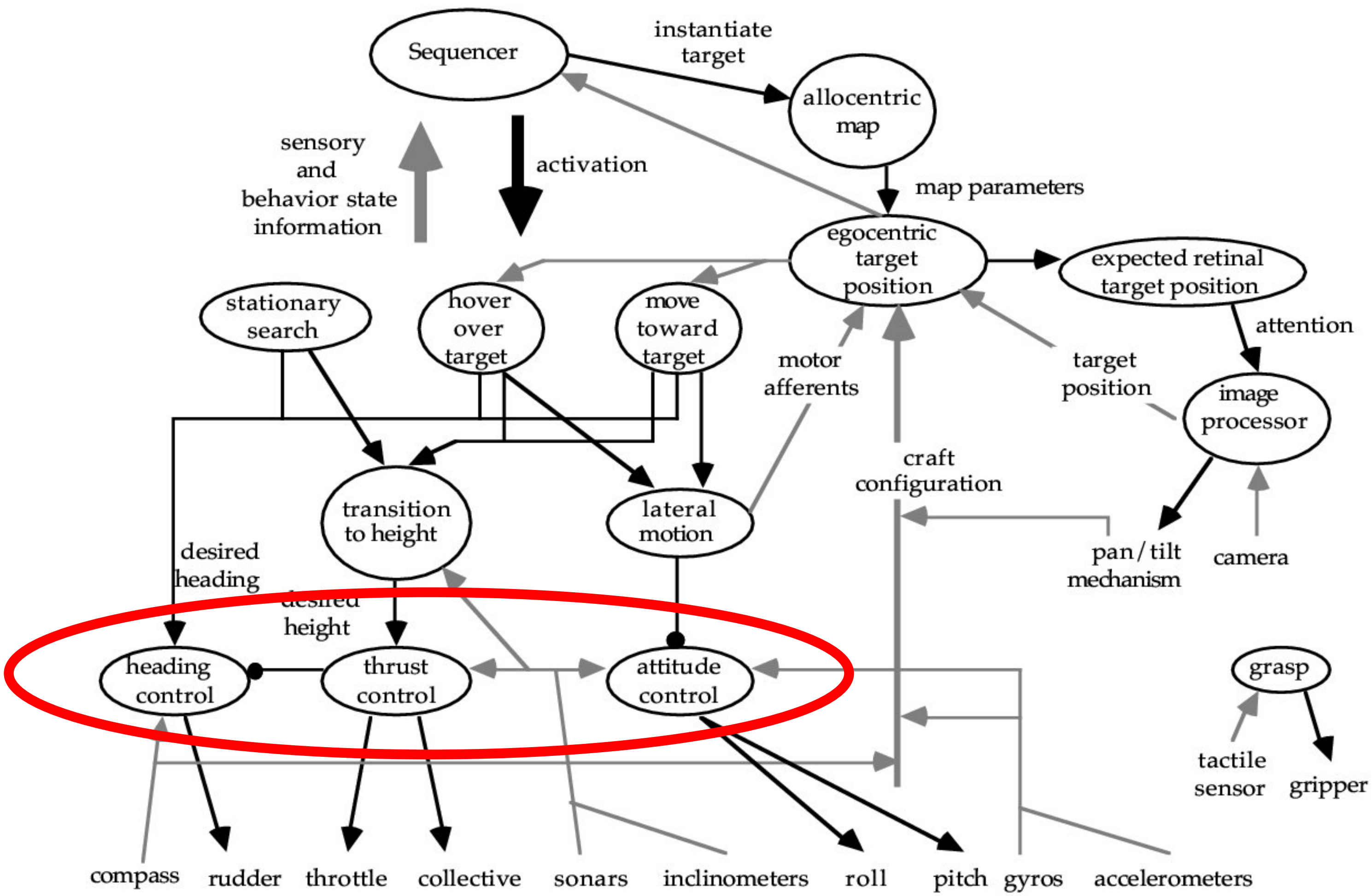
Priority-Based Scheduling

- Each process is assigned an integer **priority**
- Selecting the next process: of all the processes that are **ready**, pick the one with the highest priority

Hybrid Scheduler Example

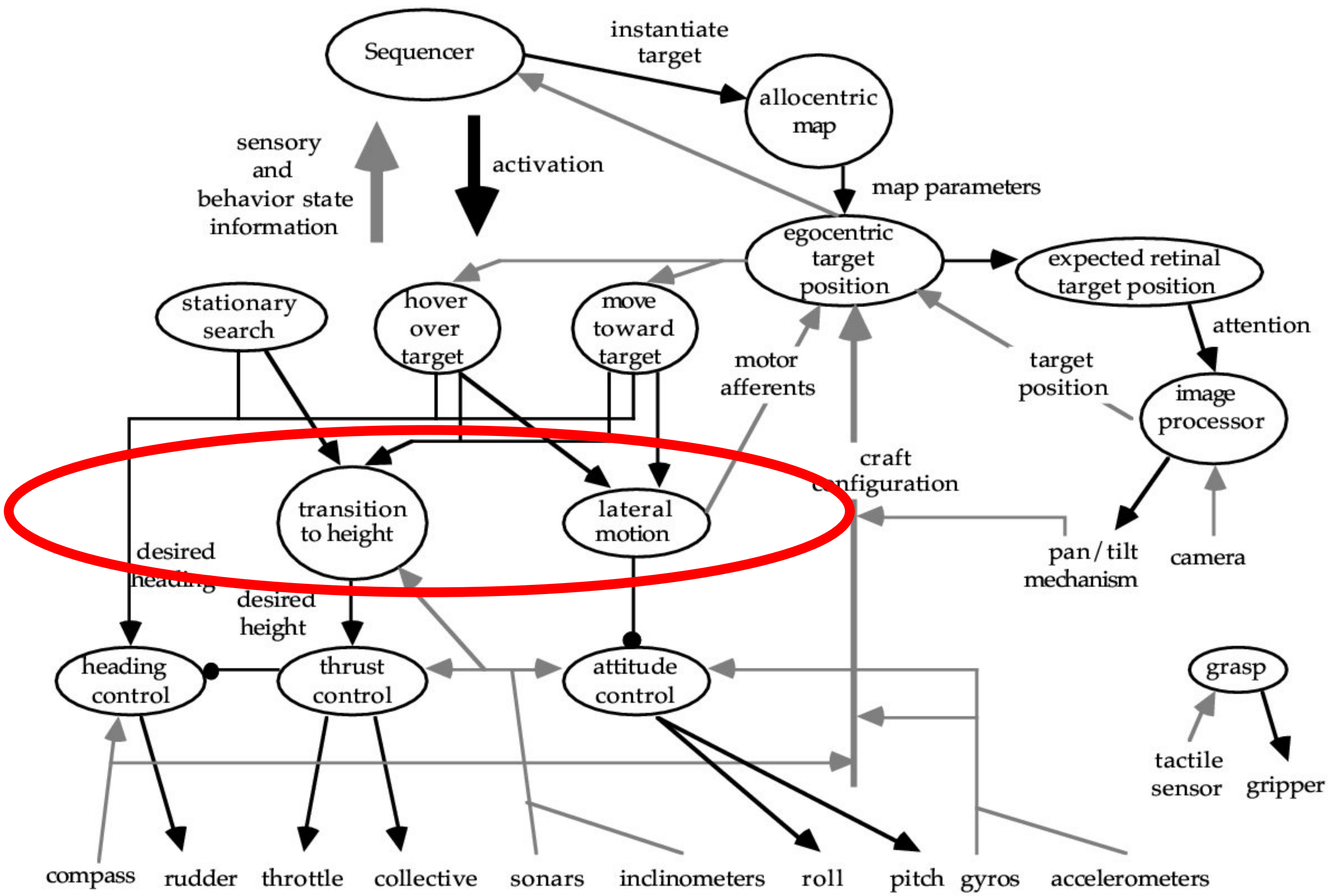
- Have a queue for each distinct priority level
- Use round robin for the highest priority queue
- If there are no processes to execute, then perform round robin between the processes in the next queue
- Repeat

Heli Example



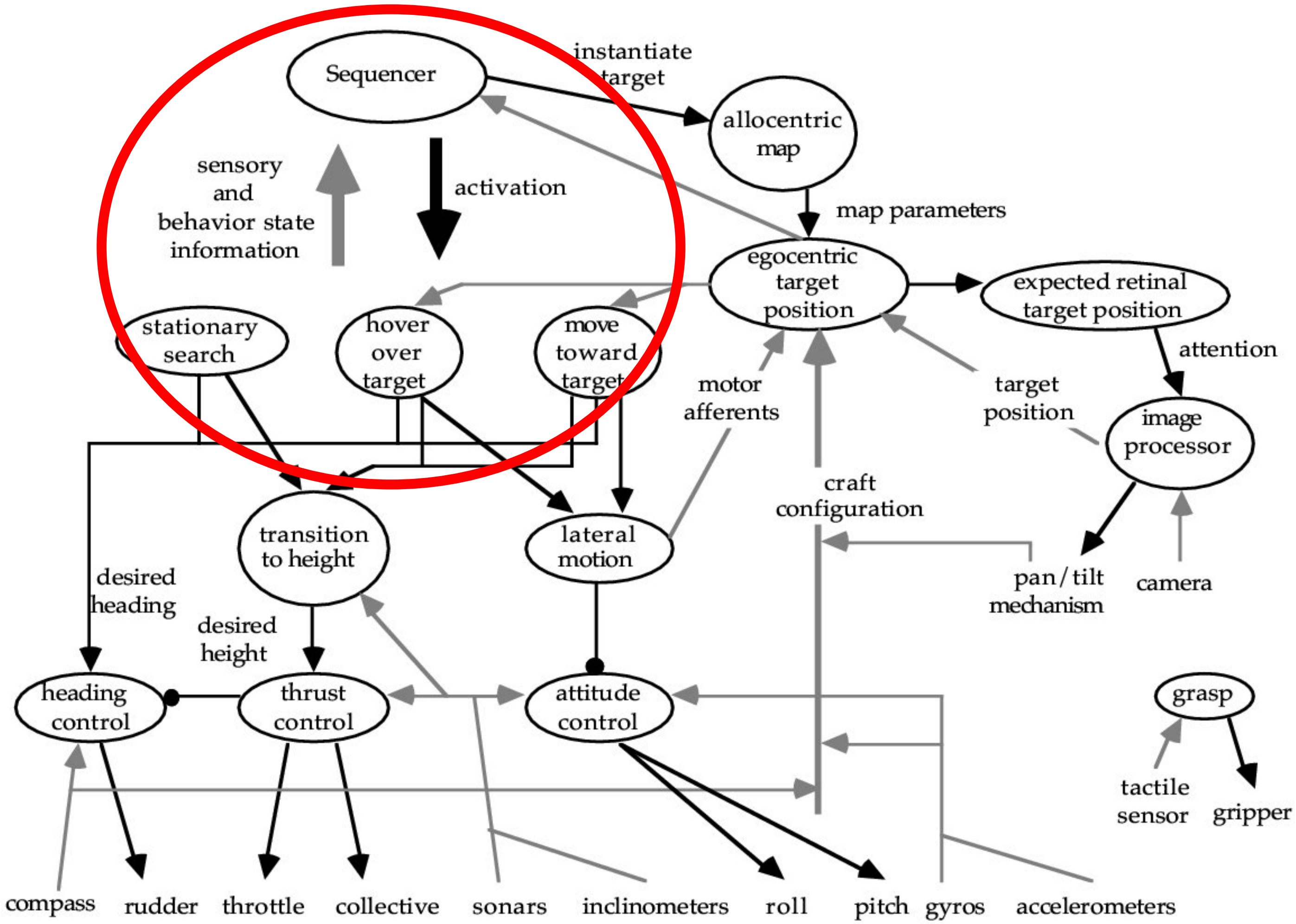
Processes with strict timing requirements are the highest priority processes

Heli Example



Many processes operate on timescales of seconds

Heli Example



Other processes operate at timescales of 10s of seconds

Shortest-Process-First Scheduling

Select the process that will execute for the shortest period of time before giving up the processor

Challenges with this?

Shortest-Process-First Scheduling

Select the process that will execute for the shortest period of time before giving up the processor

- How do we know how much time a process will take? We could:
  – Require a process to declare this
  – Estimate from past process behavior
- Can lead to starvation of low-priority processes

Non-Preemptive Scheduling

- So far, we have assumed that a process has voluntarily given up the processor
- This works if we are careful in our implementation
- But – we can have problems if a process does not “play nice”

Preemptive Scheduling

A process can be forced off the processor by the operating system

- Typically, a process is given a fixed-duration **timeslice** in which to execute
- If the process does not give up the processor within this time:
  - A different process is given the processor
  - The process is returned to the **ready** state

Hybrid Scheduler II

Combine preemption and priority-based scheduling

Hybrid Scheduler II

Combine preemption and priority-based scheduling ("priority preemptive scheduling")

- A process can be preempted at any time by a higher-priority process

Hybrid Scheduler III

Combine preemption and priority-based scheduling

- The number of timeslices that a process is given within a particular period of time is proportional to its priority

Rate Monotonic Scheduling

N tasks:

- T_i = the period between executions of task i
- E_i = worst case execution time
- So: E_i/T_i = the fraction of processor time required by task i

Scheduling Regular Tasks

In many control systems, tasks (processes) must be executed at a regular frequency

- How can we be sure that all tasks can be performed?

Rate Monotonic Scheduling

- Preemptive scheduling
- Process priority is proportional to execution frequency

RMS Theorem

- Want to know if we can execute all of our processes
- A set of processes is schedulable if:

$$\sum_i \frac{E_i}{T_i} \leq n\left(2^{1/n} - 1\right)$$

RMS Example

3 processes

- 1 ms at 100 Hz
- 10 ms at 20 Hz
- 100 ms at 2 Hz

RMS Example

3 processes

- 1 ms at 100 Hz
- 10 ms at 20 Hz
- 100 ms at 4 Hz

100 * .001 + 20 * .01 + 4 * .1 <= .7798

Yes!

RMS Example II

4 processes

- 0.1 ms at 2000 Hz
- 1.5 ms at 120 Hz
- 8 ms at 40 Hz
- 13 ms at 10 Hz

RMS Example II

4 processes

- 0.1 ms at 2000 Hz
- 1.5 ms at 120 Hz
- 8 ms at 40 Hz
- 13 ms at 10 Hz

2000 * .0001 + 120 * .0015 + 40 * .008 + 10 * .013
<= 0.7568

No!

Next Time

Coordination between processes

- Inter-process communication
- Synchronization and mutual exclusion

Last Time

- Scheduling
- Round Robin
- Priority-based
- Shorted-Process-First
- Preemption
- Rate Monotonic Scheduling

Today

- A bit more on scheduling
- Process synchronization

Administrivia

- Project 4 due today @5:00
- Homework 6 due on Tuesday @5:00

- Next Tuesday: virtual guest visit

Jim Montgomery
Robotic Software Systems Group
NASA/Jet Propulsion Laboratory

  – Readings posted on D2L

Administrivia II

- Need office hours tomorrow/Monday?
- Quizzes graded by tomorrow

- Final exam: submit example questions and answers to the D2L discussion board

Scheduling Regular Tasks

N tasks (processes):

- T_i = the period between executions of task i
- E_i = worst case execution time
- So: E_i/T_i = the fraction of processor time required by task I

Key: want a process to complete its i^{th} execution before i+1 enters the ready queue

An Example Scheduling Problem

	T_i	E_i
Process 1	100 ms	30 ms
Process 2	250 ms	40 ms
Process 3	1 s	60 ms

- All start in the ready queue at time 0
- Process 1 is first in the queue (2 is the 2nd)
- Round Robin scheduling

What is the sequence of execution?

Quiz Problem

Scheduling algorithm: priority with preemption

- Process 1: highest priority
- Process 2: middle priority
- Process 3: lowest priority

Given the same set of processes, what is the sequence of execution?

Group Quiz Problem #2

Scheduling algorithm: round robin with timeslices

- Assume 20 ms timeslices

Given the same set of processes, what is the sequence of execution?

Group Quiz Problem #3

	T_i	E_i
Process 1	50 ms	25 ms
Process 2	100 ms	40 ms

Scheduling algorithm: priority with preemption

- Assume Process 1 has highest priority
- Schedule out to 125 ms

Group Quiz Problem #4

	T_i	E_i
Process 1	50 ms	25 ms
Process 2	100 ms	40 ms

Scheduling algorithm: priority with preemption

- Assume Process 2 has highest priority
- What choice would **Rate Monotonic Scheduling** make about priority?
- Does RMS say that these processes are necessarily schedulable?

Scheduling

What did we learn?

- Priority matters!
- The Rate Monotonic Scheduling constraint is a **sufficient** condition for schedulability - but not a **necessary** one

Rate Monotonic Scheduling

	T_i	E_i
Process 1	50 ms	25 ms
Process 2	75 ms	30 ms

What is the total processor utilization?

Rate Monotonic Scheduling

	T_i	E_i
Process 1	50 ms	25 ms
Process 2	75 ms	30 ms

What is the total processor utilization?

90%

Do we pass the RMS constraint?

Rate Monotonic Scheduling

	T_i	E_i
Process 1	50 ms	25 ms
Process 2	75 ms	30 ms

Do we pass the RMS constraint?

NO

What is the schedule anyway?

Process Synchronization

So far:

- Allowing many different processes to share the same processor
- But – we have assumed that these processes are independent from one another (not generally the case)

Process Dependence

We have already seen this:

- Project 3: the interrupt routine provided data (encoder counts) that were used by the main program

- Serial buffering: the interrupt routine placed arriving bytes into a common buffer. These bytes were read out asynchronously by the main program

Process Dependence

Sharing data structures between is not always a problem …

Process Dependence

Sharing data structures between is not always a problem …

- It becomes a problem when one process can interrupt the other at an arbitrary time
  - In particular, in the middle of the modification or reading of a shared data structure

Process Dependence

Sharing data structures between is not always a problem …

- It becomes a problem when one process can interrupt the other at an arbitrary time
  - In particular, in the middle of the modification or reading of a shared data structure
  - Such as: with interrupt routines or in some form of preemptive scheduling

A Synchronization Problem

Consider the following code that you "execute":

```
if(noMilk())
{
   buyMilk();
}
```

A Synchronization Problem

Consider the following code that you “execute”:

```
if(noMilk())
{
    buyMilk();
    putMilkInFridge();
}
```

This works great, but …

A Synchronization Problem

This works great, but …

- Suppose your roommate has the same program
- What can happen?

Synchronization and Milk

Consider the following sequence of events:

Time	You	Your Roommate
3:00	Arrive at home	
3:05	Look in fridge: no milk!	
3:10	Leave for store	
3:15		
3:20	Arrive at store	
3:25	Buy milk	
3:35	Arrive at home: put milk in fridge	
3:45		
3:50		

Synchronization and Milk

Consider the following sequence of events:

Time	You	Your Roommate
3:00	Arrive at home	
3:05	Look in fridge: no milk!	
3:10	Leave for store	
3:15		Arrive at home
3:20	Arrive at store	Look in fridge: no milk!
3:25	Buy milk	Leave for store
3:35	Arrive at home: put milk in fridge	
3:45		
3:50		

Synchronization and Milk

Consider the following sequence of events:

Time	You	Your Roommate
3:00	Arrive at home	
3:05	Look in fridge: no milk!	
3:10	Leave for store	
3:15		Arrive at home
3:20	Arrive at store	Look in fridge: no milk!
3:25	Buy milk	Leave for store
3:35	Arrive at home: put milk in fridge	
3:45		Arrive at store; Buy milk
3:50		Arrive at home: put milk in fridge
		Error!

Synchronization and Milk

- The processes: you and your roommate
- The common data structure: the refrigerator
- Asynchronous access to the data structure:
  - You view the state of the fridge at a different time than when you change its state

Synchronization Concepts

- Mutual exclusion: ensure that only one process has access to a data structure at any one time (no matter how many operations it must perform)
- Synchronization: use of **atomic operators** to ensure this safe access to data structures
  - Atomic operator: cannot be interrupted

Synchronization Concepts

- Critical section: a piece of code that can only be executed by one process at a time
- Lock: (one) mechanism to ensure exclusive access
  - Lock before accessing common data structure
    - Must wait if it is already locked
  - Access the data
  - Unlock

A Solution in Code

```
lockFridge();
if(noMilk())
  {
     buyMilk();
     putMilkInFridge();
  }
unlockFridge();
```

Solution Notes

- Shared resource is locked while it is being accessed
- Must ensure that the resource is unlocked after completion

- Locking implies process waiting
  - Can improve this example with a more clever implementation

- Locking/unlocking provided by a library or the OS

Additional Questions

1. Project work contributed to understanding of the material
2. Project group was appropriately sized
3. Project group functioned well
4. I learned something (class-oriented) from a lab-mate
5. Readings were helpful in understanding class material
6. Class and book (on top of prerequisite programming class) provided appropriate background for project work