Real-Time Operating Systems (Chapter 11)

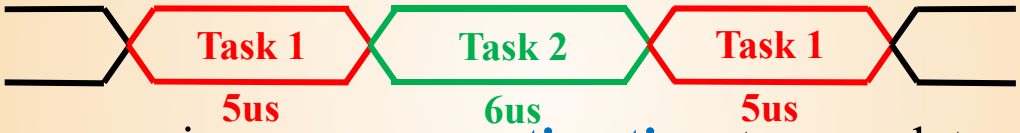
- Multitasking system solves a problem using several tasks, pieces of code that work together in an organized way.
- Coordination requires :
  - exchanging information (sharing data between Tasks)
  - synchronizing tasks
  - scheduling task execution
  - sharing resources
- Software that does these things is called an **operating system**
- **An operating system that can meet specified time constraints** for task execution is called a **real-time operating system – RTOS.**

Tasks

- Task are like the individual jobs that must be done (in coordination) for a large job.
- We can partition the design based on things that we think can/should be done together, or just in a way to make the problem easier to think about, or based on knowing the most efficient partitioning for execution.
- Example tasks/design partitions for a digital thermometer with flashing temperature indicator:
  - Detect & signal button pressed
  - Read temperature & update flash rate
  - Update LCD
  - Flash LED

Tasks/Processes (cont)

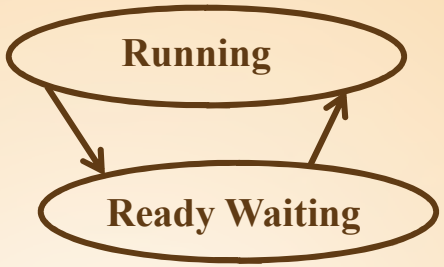
- Tasks/processes require resources or access to resources
  - processor (for execution)
  - stack
  - memory
  - registers
  - P.C.
  - I/O Ports
  - network connections
  - file descriptors
  - etc..
- These are allocated to a process when it is executed by the operating system.
- The contents of the P.C. & other pieces of data associated with the task determine its process state.

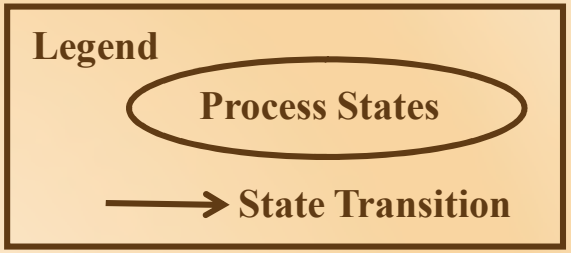


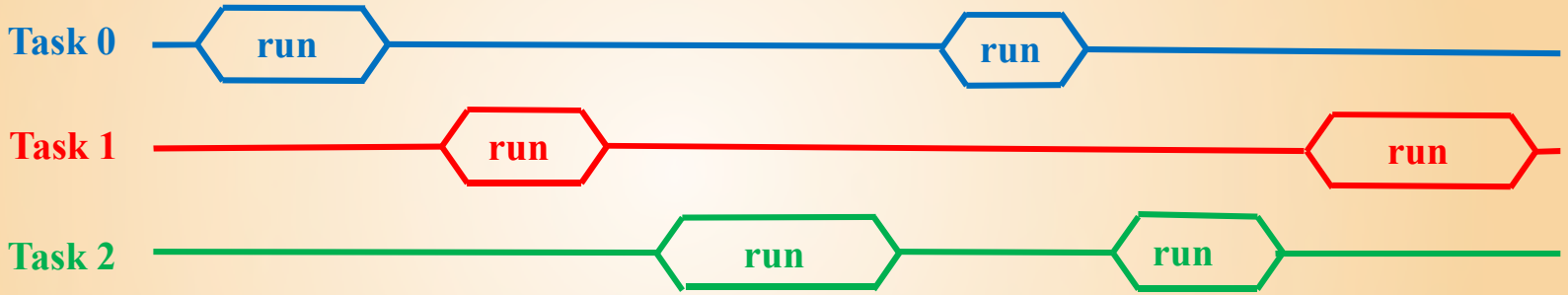
- Each process requires some **execution time** to complete. **Task 1 = 10us**
- We'll call the time between the start and termination of a task its **persistence.** **Task 1 = 16us**
- Since several tasks time-share the CPU and other resources, execution time may not equal the persistence.
- The OS manages resources, including CPU time, in slices
- to create the effect of several tasks executing concurrently, seemingly performing their jobs at the same time. Though, in reality they may not be running at the same instant in time unless you have a multi-threaded or multi-core processor.

Scheduling

- The illusion of concurrent execution is created by scheduling a scheduling process the moves tasks between states.



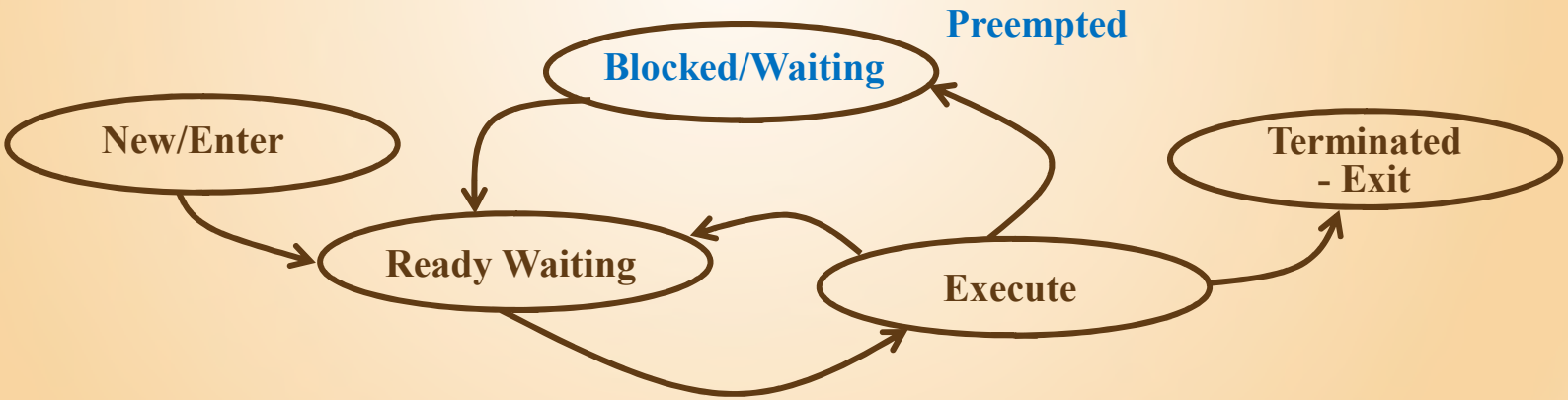




- Options for Scheduling Strategies:
  - Multiprogramming: tasks run until finished or until they must wait for a resource, e.g. I/O
  - Real-time:Tasks are scheduled and guaranteed to complete with strict timing specified
  - Time-sharing, tasks are interrupted, or preempted, after specified time slices to allow time for other tasks to execute

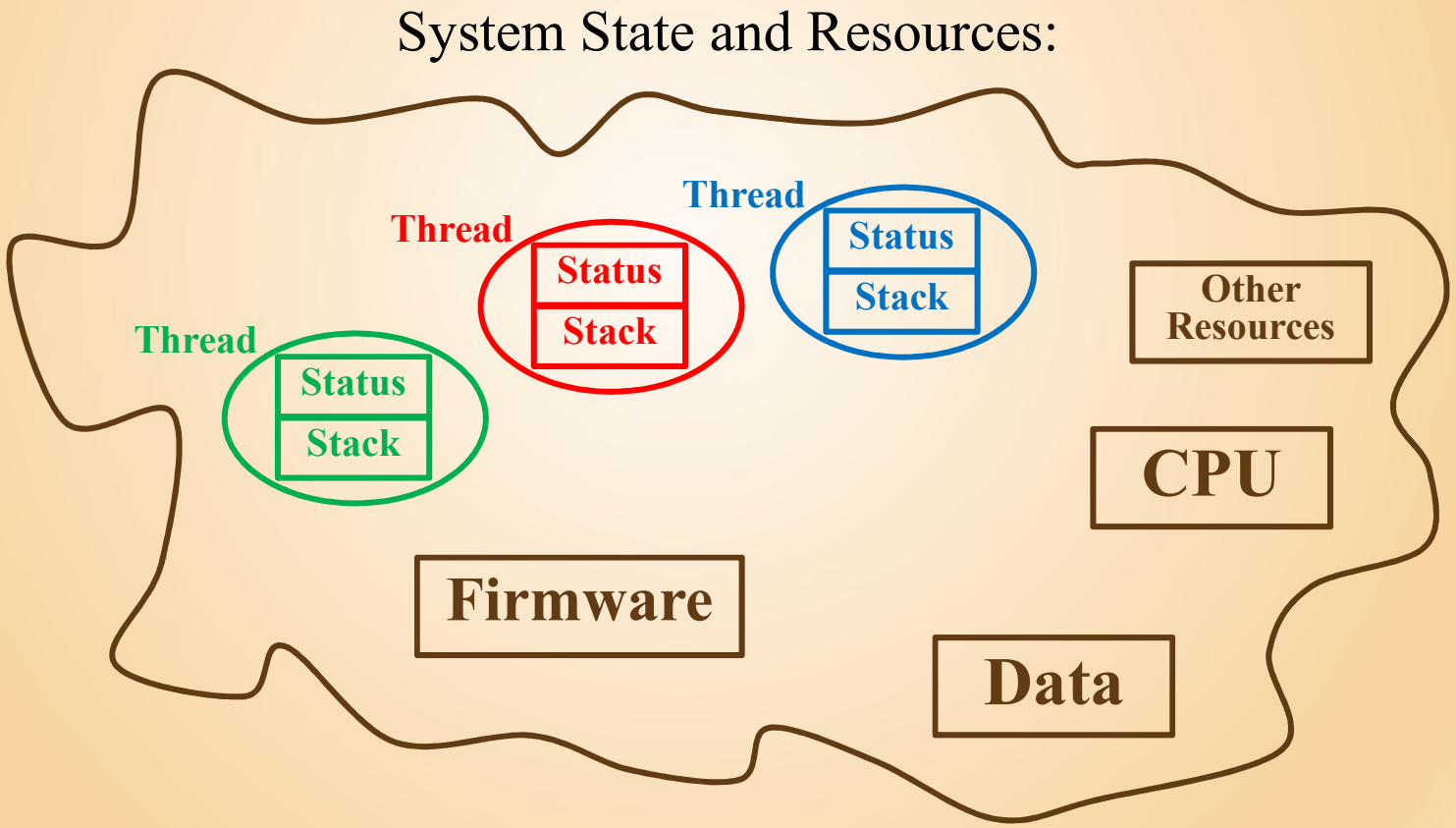
Preempting/Blocking

- **Preempting/blocking** requires saving the state of the process as it is running in the processor, called the context, including P.C. and registers, so that the process can be stopped/blocked and the context can be restored later for the process to resume just as it left off. (similar to saving state when a function is called and restoring state at the end of a function)
- This saving of state of one process and loading of another is called context switching. It is the overhead of multitasking.



Threads

- Think of a thread as an organizational concept that is the smallest set of (information about) resources required to run a program, including a copy of the CPU registers, stack, PC. The OS manages several tasks formally as threads.



Threads

- Ideally, each process should have its own private section of memory to work with, called its address space.
- Along with hardware support (memory protection unit MPU), an OS may be able to enforce that processes do not access memory outside their own address space.
- Organizational concepts (may have one or both):
  - Multi-process execution: multiple distinct processes running in separate threads
  - Multi-threaded process: a process with several execution threads (likely sharing memory and managing resource use internally)
- Note: **intra**process thread context switching is typically less expensive than **inter**process context switching.

Reentrant Code & Thread Safe Code

- By default, all code is not safe to run alongside other code "simultaneously" or even along side itself

- a function with:

thread safe code means other threads or processes can run safely the same time	(safety with respect to other code)
Reentrant code handles multiple simultaneous calls	(safety with respect to same code)

Thread safe but not reentrant Example:

- To allow multiple processes to safely time-share a resource, an OS typically provides check, lock, and free utility functions. These are used to make a code thread safe. †

```
int AFunction() {

  //some function that checks and waits for
  // availability of a resources and locks/reserves
  // it so other processes won't access it
  // -> makes this thread safe

  wait_for_free_resource_and_then_lock_access();

  do_some_stuff();

  //free/unreserved the resource
  unlock_some_resource();
}
```

- But is this code always reentrant?

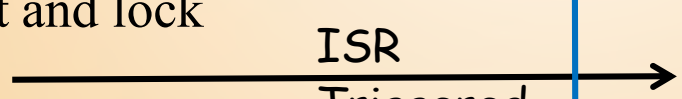
†These types of utility functions are called mutex (mutually exclusive) functions and are provided by the OS; we'll discuss mutex functions more later.

Thread safe but not reentrant Example: (cont)

- Function Synopsis:
  - Wait and lock
  - Use
  - Free

- To look at reentrant lets look at simultaneous calls from a main thread and an ISR can call over and over in the same process while function is still running, but if another thread deletes the file then from the other's threads perspective the file handle becomes bad without any warning while it is using it.

- Main calling `Afunction`:
  - Wait and lock
  - Use
  - Free



- ISR calling `Afunction`:
  - Wait and lock
  - Use
  - Free

Stuck here, the resource is locked, Never to reach return to free resource

Thread safe but not reentrant Example: (cont)

- To prevent this, we need a way for a process to "lock" a resource to be able to hold its assumptions.

```
int function() {

  char *filename="/etc/config";
  FILE *config;

  if(file_exist(filename)){
    // what if file is deleted by another
    //    process at this point?

    config=fopen(filename,"r"); //At this point, many OSs
                                //will prevent deletion

    ...use file here..

    }
  }
```

- Pasted from http://en.wikipedia.org/wiki/Thread_safety

Reentrant and Thread Safe Coding Practices

- Dangerous -multiple calls access the same variable/resource
  - global variables
  - process variables
  - pass-by-reference parameters
  - shared resources
- Safe
  - local variables -only using local variables makes code reentrant by giving each call its own copy
- For example, some string functions like strtok() † use global variables and are not reentrant
  - † The C library function char *strtok(char *str, const char *delim) breaks string str into a series of tokens using the delimitrer delim.
  - http://www.tutorialspoint.com/c_standard_library/c_function_strtok.htm

Kernel

- The "core" OS functions are as follows:
  - perform scheduling -> handle by the "scheduler"
  - dispatch of processes -> handle by the "dispatcher"
  - facilitate inter-process communication
- A kernel is the smallest portion of OS providing these functions

Functions of Operating System

- Process or Task Management
  - Process creation, deletion, suspension, resumption
  - Management of interprocess communication
  - Management of deadlocks (processes locked waiting for resources)
- Memory Management
  - Tracking and control of tasks loaded in memory
  - Monitoring which parts of memory are used and by which process
  - Administering dynamic memory allocation if it is used
- I/O System Management
  - Manage Access to I/O
  - Provide framework for consistent calling interface to I/O devices utilizing device drivers conforming to some standard
- File System Management
  - File creation, deletions, access
  - Other storage maintenance
- System Protection
  - Restrict access to certain resources based on privilege
- Networking: -for distributed applications,
  - Facilitates remote scheduling of tasks
  - Provides interprocess communications across a network
- Command Interpretation
  - Accessing I/O devices through devices drivers, interface with user to accept and interpret command and dispatch tasks

RTOS

- An RTOS follows (rigid) time constraints. Its key defining trait is the **predictability**(repeatability) of the operation of the system, **not speed**.
  - hard-real time -> delays known or bounded
  - soft-real time -> at least allows critical tasks to have priority over other tasks

- Some key traits to look for when selecting an OS:
  - scheduling algorithms supported
  - device driver frameworks
  - inter-process communication methods and control
  - preempting (time-based)
  - separate process address space
  - memory protection
  - memory footprint, data esp. (RAM) but also its program size (ROM)
  - timing precision
  - debugging and tracing

Task Control Block

- The OS must keep track of each task. For this, a structure called as task control block(TCB) or a process control block can be used. They will be stored in a Job Queue implemented with pointers (array or linked-list).

- A prototypical TCB, it's generic to support a variety of functions.

```
struct TCB {
  void(*taskPtr)(void *taskDataPtr); //task function(pointer),one arg.
  void *taskDataPtr;                 // pointer for data passing
  void *stackPtr;                    // individual task's stack
  unsigned short priority;           // priority info
  struct TCB * nextPtr;              // for use in linked list
  struct TCB * prevPtr;              // for use in linked list
}
```

- Think about the aspect of being generic. A task can be just about anything that a computer can do. A generic template must be used to handle any task.

- Each task is written as a function conforming to a generic interface:

```
void aTask( void * taskDataPtr){
  // this task's code
}
```

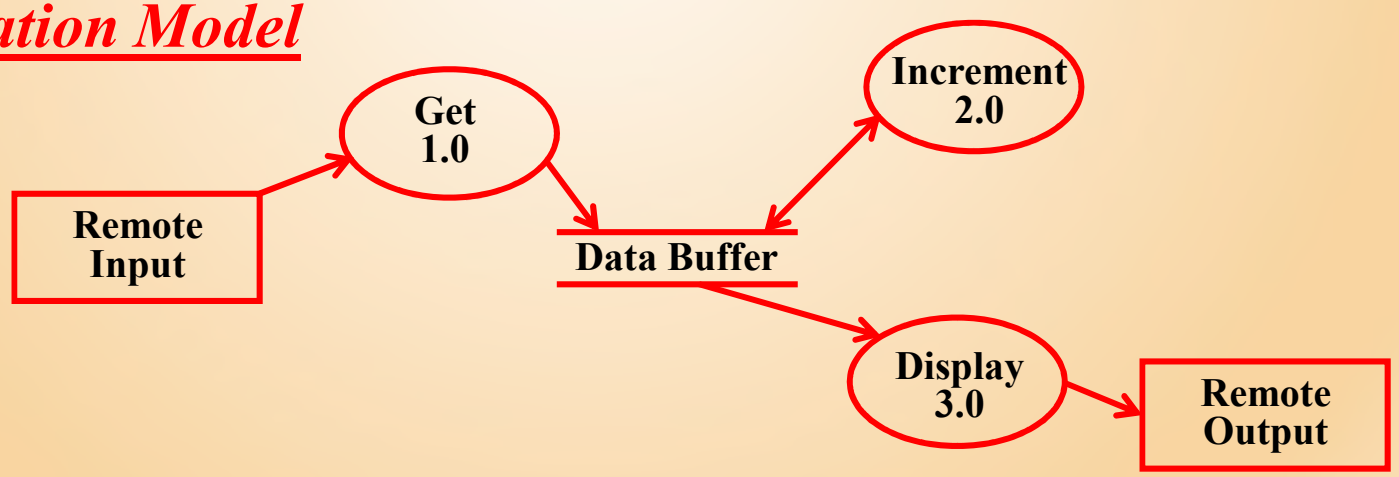
- Each task's data is stored in a customized container. The task must know the structure, but the OS only refers to it with a generic pointer.

```
struct taskData{
  int  task Data0;
  int  task Data1;
  char task Data2;
}
```

A simple kernel example

- Tasks to be performed for this example:
  - Bring in some data
  - Perform computation on the data
  - Display the data
- First Implementation – Step 1:
  - System will run forever cycling through each task calling the task and letting it finish before moving on
- Second Implementation – Step 2:
  - Declares a TCB for each task
  - TCB contains a function pointer for the task
  - Data to be passed to the task
  - Task queue implemented using array, each task runs to completion
- Third Implementation – Step 3:
  - Adds usage of ISR to avoid waiting

Implementation Model



First Implementation – Step 1: - will run forever

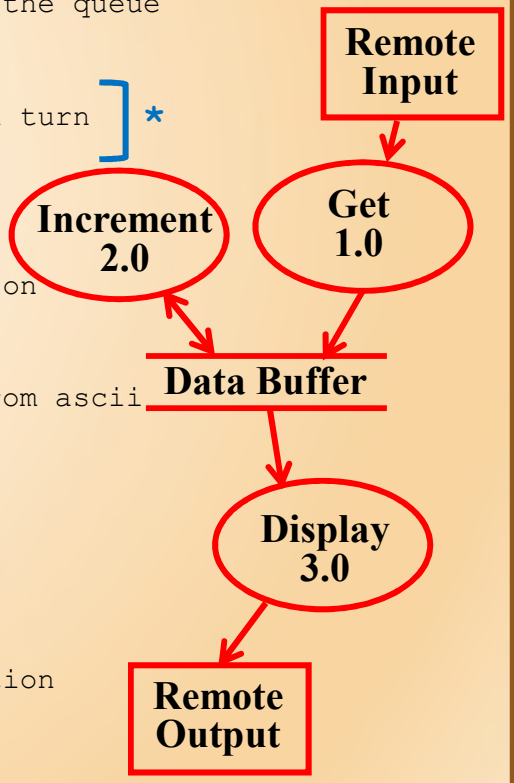
```
#include <stdio.h> // Building a simple OS kernel -step 1
// Declare the prototypes for the tasks

void get (void* aNumber);          // input task
void increment (void* aNumber);    // computation task
void display (void* aNumber);      // output task

void main(void) {
  int i=0;                         // queue index
  int data;                        // declare a shared data
  int* aPtr = &data;               // point to it
  void (*queue[3])(void*);         // declare queue as an array of pointers to
                                   //   functions taking an arg of type void*
  queue[0] = get;                  // enter the tasks into the queue
  queue[1] = increment;
  queue[2] = display;
  while(1) {
    queue[i] ((void*) aPtr);       // dispatch each task in turn
    i = (i+1)%3;
  }
  return;
}

void get (void* aNumber) {         // perform input operation
  printf ("Enter a number: 0..9 ");
  *(int*) aNumber = getchar();
  getchar();                       // discard cr
  *(int*) aNumber -= '0';          // convert to decimal from ascii
  return;
}

void increment (void* aNumber) { // perform computation
  int* aPtr = (int*) aNumber;
  (*aPtr)++;
  return;
}

void display (void* aNumber) {   // perform output operation
  printf ("The result is: %d\n", *(int*)aNumber); return;
}
```

Get input

Do Computation

Display

-System will run forever cycling through each task calling the task and letting it finish before moving on

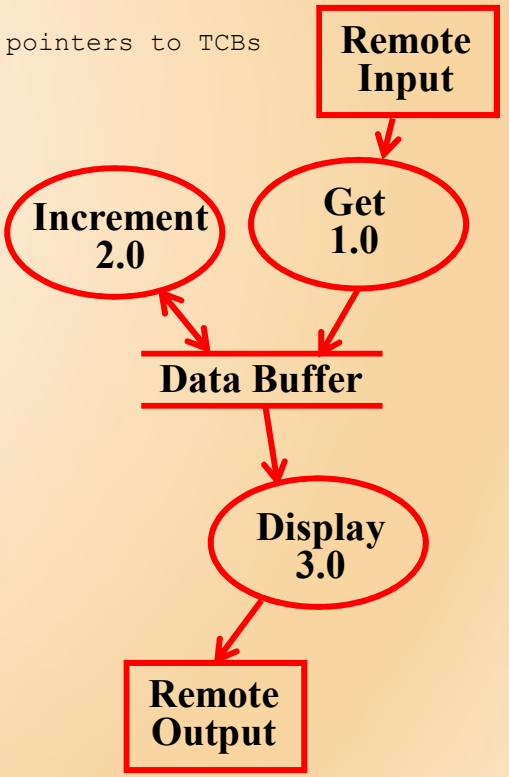


Second Implementation – Step 2 increased tasks in main – (code continues on next slide):

```
#include <stdio.h> // Building a simple OS kernel -step 2
                   // Declare the prototypes for the tasks

void get (void* aNumber);       // input task
void increment (void* aNumber); // computation task
void display (void* aNumber);   // output task

// Declare a TCB structure
typedef struct {
  void* taskDataPtr;
  void (*taskPtr)(void*);
} TCB;

void main(void) {
  int i=0;                      // queue index
  int data;                     // declare a shared data
  int* aPtr = &data;            // point to it
  TCB* queue[3];                // declare queue as an array of pointers to TCBs

  // Declare some TCBs
  TCB inTask, compTask, outTask;
  TCB* aTCBPtr;

  // initializetheTCBs
  inTask.taskDataPtr = (void*)&data;
  inTask.taskPtr = get;
  compTask.taskDataPtr = (void*)&data;
  compTask.taskPtr = increment;
  outTask.taskDataPtr = (void*)&data;
  outTask.taskPtr = display;

  // Initialize the task queue
  queue[0] = &inTask;
  queue[1] = &compTask;
  queue[2] = &outTask;

  // schedule and dispatch the tasks
  while(1) {
    aTCBPtr = queue[i];
    aTCBPtr->taskPtr((aTCBPtr->taskDataPtr));
    i = (i+1)%3;
  }

  return;
}
```

-Declares a TCB for each task
-TCB contains a function pointer for the task
-Data to be passed to the task
-Task queue implemented using array, each task runs to completion

Defines a 'template' for defining functions, signatures



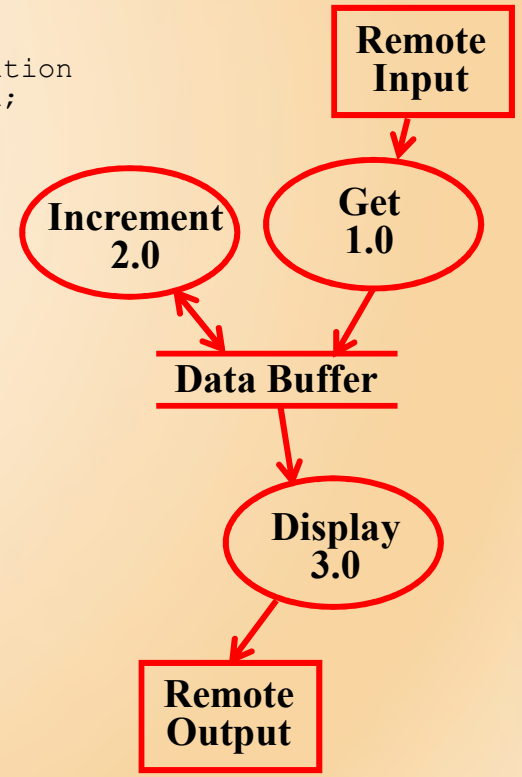
Second Implementation – Step 2 (cont.) – Follow on functions – same as Step 1…

Get input

Do Computation

Display

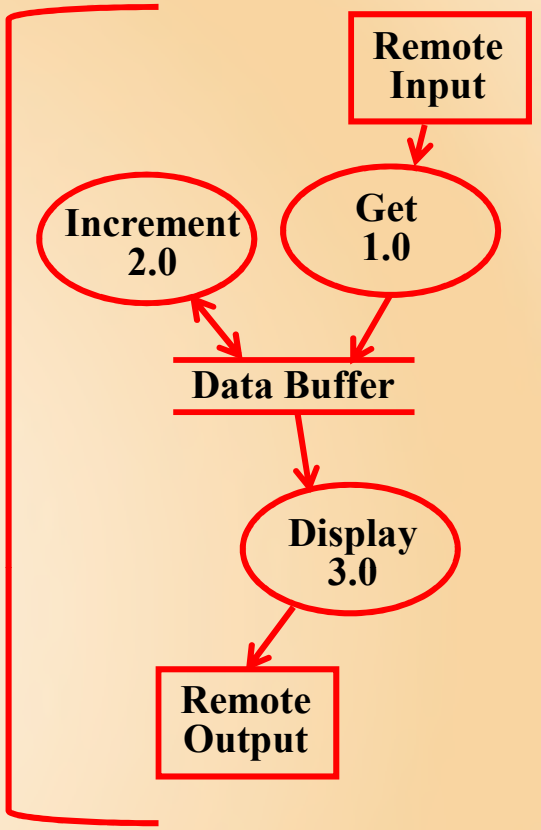
```
void get (void* aNumber) {          // perform input operation
  printf ("Enter a number: 0..9 ");
  *(int*) aNumber = getchar();
  getchar();                        // discard cr
  *(int*) aNumber -= '0';           // convert to decimal from ascii
  return;
}

void increment (void* aNumber) { // perform computation
  int* aPtr = (int*) aNumber;
  (*aPtr)++;
  return;
}

void display (void* aNumber) {   // perform output operation
  printf ("The result is: %d\n", *(int*)aNumber); return;
}
```



ISR Suggestion:

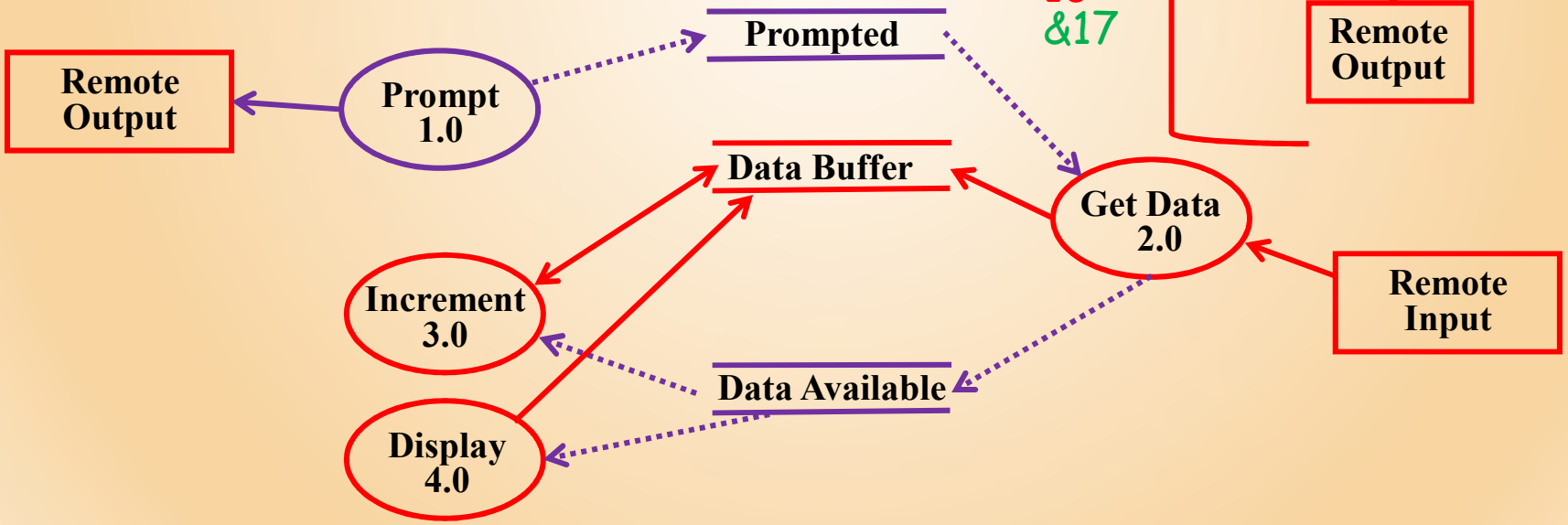
- Problems:
  - If any task must wait for something, no other task can run until the running task no longer needs to wait. This can lead to system "hanging", trivially waiting on something
  - In this case, no updates can happen while waiting on user input
  - Would be better to break task up into two parts:
    1) task: display prompt
    2) task: check if user entered data and move on otherwise …. Practically implemented using interrupts

- Need ISR:



Ref: from slides 16 &17

ISR Approach:



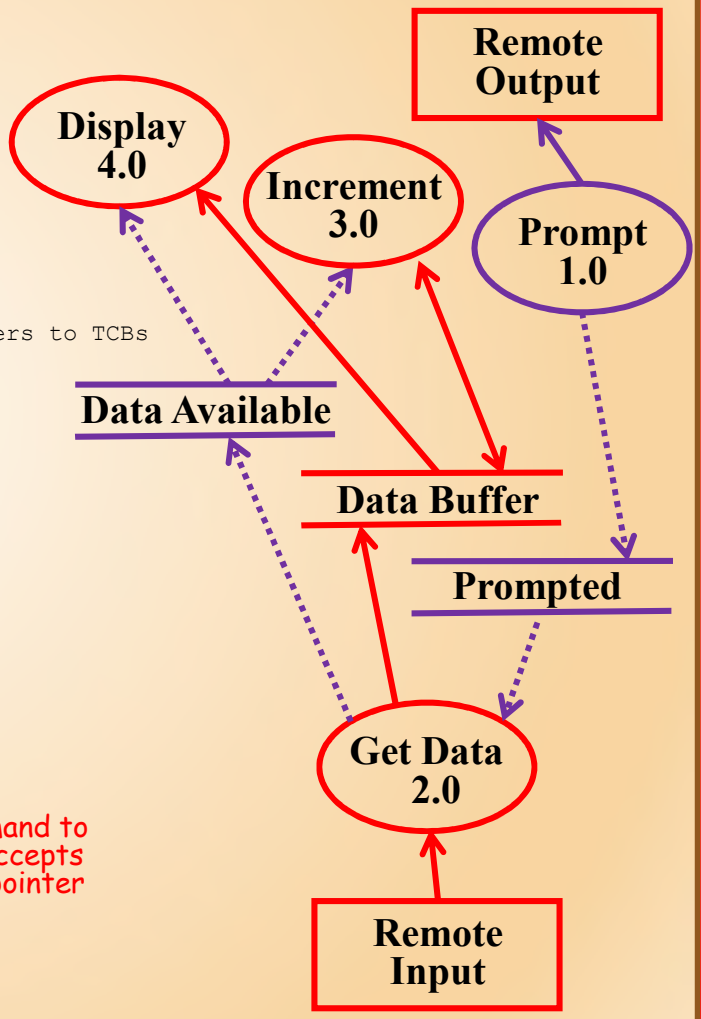
Third Implementation – Step 3 Adding in ISR (code continues on next slide):

-Adds usage of ISR to avoid waiting

```
#include <stdio.h>                // Building a simple OS kernel -step 3
#define KBINT 0x3;                // the kb will interrupt on interrupt 3
typedef enum aBool{FALSE, TRUE}; // create a boolean value typedef
unsigned char boolean;            // create a Boolean type
// Declare the globals
boolean prompted = FALSE;         // user data requested
boolean dataAvail = FALSE;        // user data available
int data;                         // declare a shared data
// Declare the prototypes for the tasks
void prompt(void);                // prompt task
void increment (void);            // computation task
void display (void);              // output task
// Declare the prototype for the ISR
void getDataISR (void);           // get data ISR
// Declare a TCB structure
typedef struct {
  void (*taskPtr)(void);
} TCB;

void main(void) {
  int i=0;                        // queue index
  TCB* queue[3];                  // declare queue as an array of pointers to TCBs

  // Declare some TCBs
  TCB promptTask;
  TCB compTask;
  TCB outTask;

  // Declare a working TCB pointer
  TCB* aTCBPtr;

  // Initialize the TCBs
  promptTask.taskPtr = prompt;
  compTask.taskPtr = increment;
  outTask.taskPtr = display;

  // Initialize the task queue
  queue[0] = &promptTask;
  queue[1] = &compTask;
  queue[2] = &outTask;

  // Enter the keyboard ISR into the interrupt vector table
  setVect(KBINT, getDataISR);

  // Schedule and dispatch the tasks
  while(1) {
    aTCBPtr = queue[i];
    aTCBPtr->taskPtr();
    i = (i+1)%3;
  }
  return;
}
```

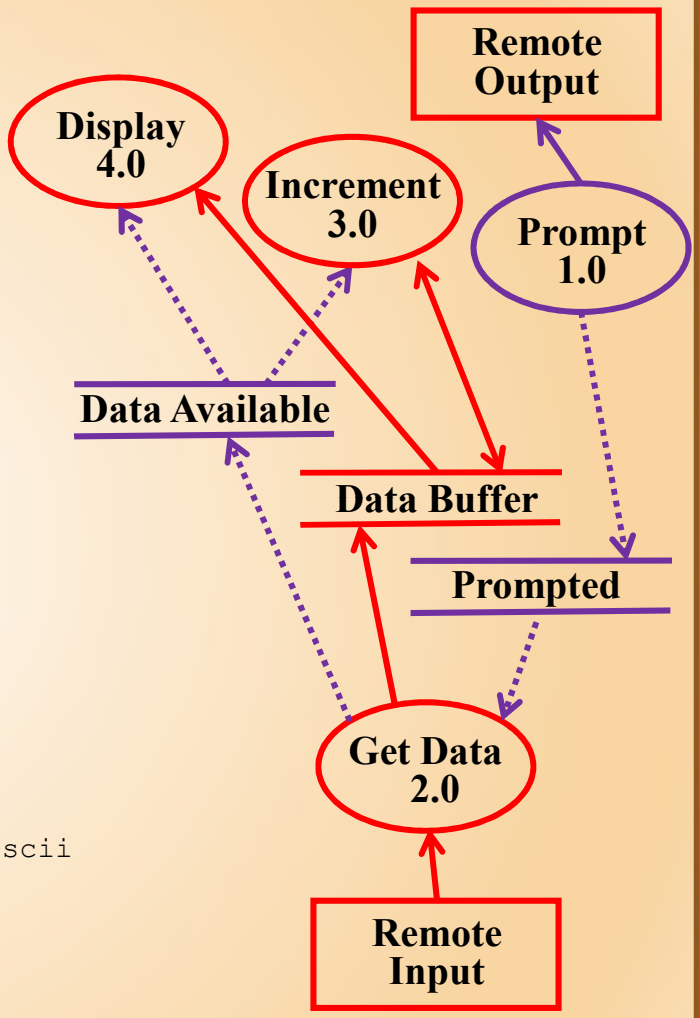
Defines template for function call

Some platform-dependent command to set ISR vector (in this case it accepts the interrupt # and a function pointer



Third Implementation – Step 3 (cont.) – Follow on functions different from first two…

```
void prompt(void) {      // perform input operation
  if (!prompted) {
    printf ("Enter a number: 0..9 ");
    prompted = TRUE;
  }
  return;
}

void increment (void) { // perform computation
  if (dataAvail) {
    data++;
  }
  return;
}

void display (void) {    // perform output operation
  if (dataAvail) {
    printf ("The result is: %d\n", data);
    prompted = FALSE;
    dataAvail = FALSE;
  }
  return;
}

// keyboard ISR
void getDataISR(void) { // perform input operation
  data = getchar();
  getchar();             // discard cr
  data -= '0';           // convert to decimal from ascii
  dataAvail = TRUE;
  return;
}
```

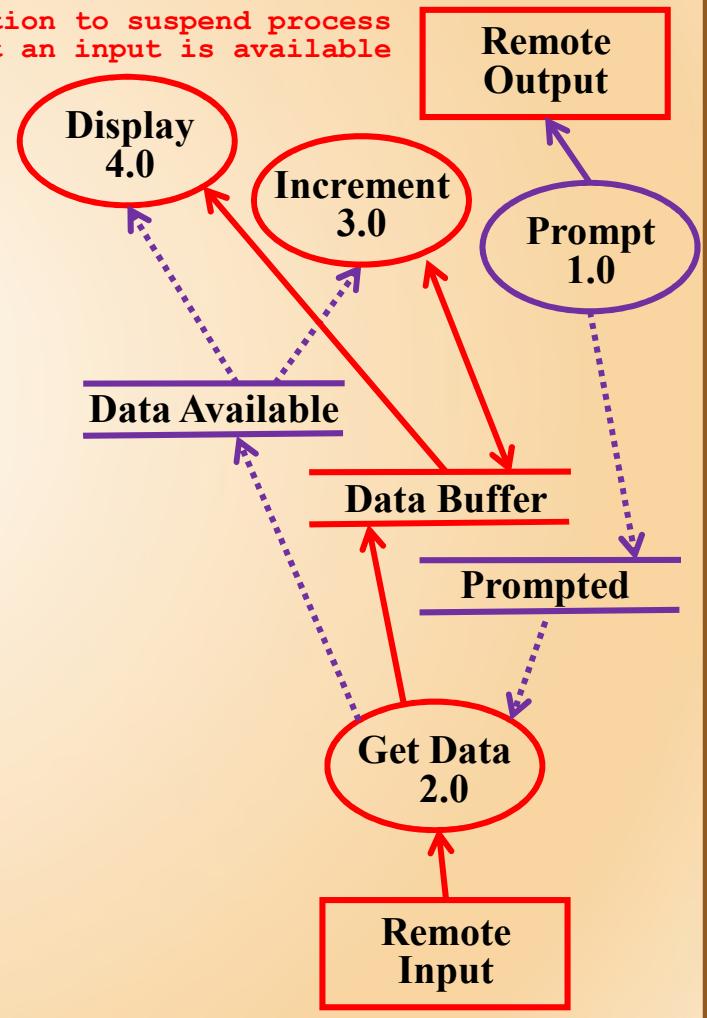


Observations/ Thoughts/Questions:

- Notice that prompt function is called forever even though it only acts once per input
  -> Would be better if prompt function could signal that it should be taken out of the queue when it is finished and reenter only when needed again
- Rather than calling every task function every time through, it would be nice if a task has a way to tell the scheduler to "time out" itself for a while (take itself temporality out of calling or execution queue (ready-wait).
- Alternatively, a task could tell the kernel to only call it again upon some condition being set, by defining a software flag and waiting for it to be set by another task.
- An even more advanced and useful feature would be if there was a call that a task could make put itself to sleep in the middle of the code and resume where it left off. (next slide)
- Thought question: What are the options for you handling a user prompt longer than 1 char?
- Most of these are addressed/enabled by the operating system

Third Implementation – Step 3 OS function suspending process until input is available:

```
void prompt(void) {      // perform input operation
  if (!prompted) {
    printf ("Enter a number: 0..9 ");
    prompted = TRUE;
    wait_for_inputAvail(); // potential OS-provided function to suspend process
                           //    until the condition that an input is available
    ch=getchar();
    data -= '0';         // convert to decimal from ascii
  }
  return;
}

void increment (void) { // perform computation
  if (dataAvail) {
    data++;
  }
  return;
}

void display (void) {    // perform output operation
  if (dataAvail) {
    printf ("The result is: %d\n", data);
    prompted = FALSE;
    dataAvail = FALSE;
  }
  return;
}

// keyboard ISR
void getDataISR(void) { // perform input operation
  data = getchar();
  getchar();             // discard cr
  data -= '0';           // convert to decimal from ascii
  dataAvail = TRUE;
  return;
}
```



Context Switching

- Involves →saving existing context
  →switching to new one
  →restoring old

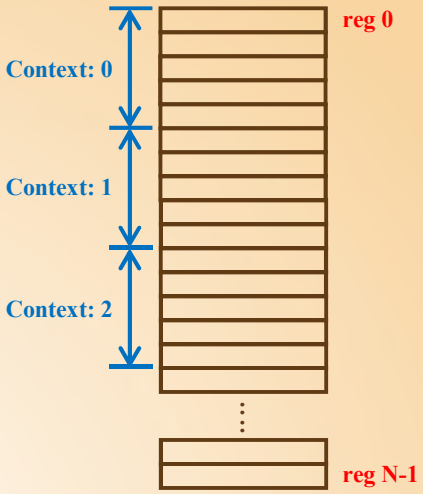
- Typical elements of context:
  - state of CPU registers flags
  - values of local variables (the stack)
  - any other status information

- Options for storage of multiple contexts:
  - use duplicate hardware-divvy general purpose registers →fast switching
  - store using task control blocks → flexible option for simple kernels and full-featured OS
  - stacks → store context in the stack along with the good for interrupt-only, foreground/background task systems or state-machine based systems

Duplicate Hardware Context

- divvy the registers among contexts (can reuse some registers for more than one context if desired)

FAST !!



Task Control Blocks

- As discussed, context stored in "struct"s.
- Tasks are managed via "pointer queues" .
  - running, waiting, etc....
- Versatile, used on full-featured OSs and simple kernels

Using Stacks

- Similar to TCB, but the context is instead stored on stack along with local variables and other typical stack data in a stack frame
- Multiple stack frames, often of some predetermined size, then live in the stack with one active at a time.
- Active stack is designated by moving the stack counter
- Stack frame contents can be managed by each process and/or foreground processes

Materials to Review for Quiz 4:

class: 18, slide: 1 → ptrplusplus.c
class: 18, slide: 2-7 → Memory problem examples
class: 18, slide: 11-12 → Interrupts - Overview, Sources
class: 18, slide: 15 → Interrupts - Vocabulary and Concepts
class: 19, slide: 4 → Coding Interrupts in AVR
class: 19, slide: 8 → Atomic Resource Access
class: 19, slide: 12 → The Interrupt Sequence for AVR
class: 19, slide: 14 → Questions and considerations for ISR coding
class: 19, slide: 23 → Getting the desired timing
class: 20, slide: 1 → Timing with the Overflow Bit
class: 20, slide: 4 → Multi-word writes
class: 20, slide: 7 → Slip Adjustment
class: 20, slide: 8 → Event Counters
class: 21, slide: 3-4 → ADC Aliasing, Sample-Hold
class: 21, slide: 5-7 → ADC Converters
class: 21, slide: 18-19 → Signed vs. Unsigned; Casting
class: 21, slide: 21-22 → Sign Extension
class: 21, slide: 34-37 → Mulitplying and Dividing with Shift
class: 22, slide: 1 → Definition of RTOS, Tasks
class: 22, slide: 2 → Problem related to Execution, Persistence
class: 22, slide: 4 → Preempting/Blocking Diagram
class: 22, slide: 5-6 → Threads
class: 22, slide: 11 → Reentrant and Tread Safe Coding Practices
class: 22, slide: 13 → Functions of Operating System
class: 22, slide: 25 → Context Switching – changed from posted slide #...