Real-Time Operating Systems

RTOS – Multitasking on embedded platforms

Real Time Operating Systems

- Operating systems - Solving problems using organized tasks that work together
- Coordination requires
  - Sharing data
  - Synchronization
  - Scheduling
  - Sharing resources
- An operating system that meets specified time constraints is called a Real-Time Operating System (RTOS)

Tasks

- Individual jobs that must be done (in coordination) to complete a large job
- Partition design:
  - Based on things that could/should be done together
  - In a way to make the problem easier
  - Based on knowing the most efficient partitioning for execution
- Example tasks/design partitions for a digital thermometer with flashing temperature indicator
  - Detect & Signal button press
  - Read Temperature & update flash rate
  - Update LCD
  - Flash LED

Tasks/Processes

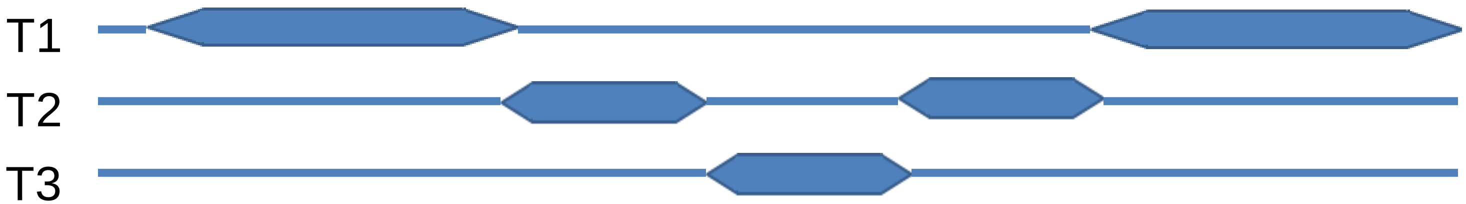
- Tasks require resources or access to resources
  - Processor, stack memory registers, P.C. I/O Ports, network connections, file, etc…
- These should be allocated to a processes when chosen to be executed by the operating system
- Contents of PC & other pieces of data associated with the task determine its process state

Task Terminology

- Execution Time – Amount of time each process requires to complete
- Persistence – Amount of time between start and termination of a task
- Several tasks time-share the CPU and other resources, execution time may not equal persistence
    - Ex. Task execution time = 10ms, is interrupted for 6ms during the middle, persistence = 16ms
- OS manages resources, including CPU time, in slices to create the effect of several tasks executing concurrently
    - Cannot operate truly concurrently unless there is a multi-core processor

Scheduling

- Illusion of concurrent execution can be created by scheduling a process that move tasks between states



- Options for scheduling strategies
  - Multiprogramming – tasks run until finished or until they must wait for a resource
  - 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

- To preempt a task:
  - Save the state of the process – called the context – including P.C. and registers
- This allows the preempting process to execute and then restore the preempted task
- Saving of state of one process and loading another is called “context switching” and is the overhead of multitasking

Threads

- 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
  - 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 process do not access memory outside their address space
- Organizational Concepts
  - 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 – intraprocess thread context switching is typically less expensive than interprocess context switching

Reentrant and Thread Safe Code

- By default all code is not safe to run alongside other code “simultaneously” or even alongside itself
- Thread safe code – other threads or processes can run safely at the same time (safety with respect to other code)
- Reentrant code – handles multiple simultaneous calls (safety with respect to same code)

Example

- To allow multiple processes to safely time-share a resource, an OS typically provides check, lock, and free utility functions.

```
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/unreserve the resource
    unlock_some_resource();
}
```

- This code may not be reentrant

Example – Not Reentrant

- Consider when there are simultaneous calls from a main thread and an ISR

Main Thread (in aFunction):	ISR Thread:
	Wait and Lock
Wait and Lock	//Stuck here waiting for resource to unlock
Use	
//Interrupted in Use	Use
//Has not freed resource	
Free	Free

Example- Not Thread Safe

```
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..
    }
}
```

- This code can be called over and over in the same process
- What if another thread deletes the file after the handle has been verified but before it has been used?
  - Creates a segfault with no way to detect while using it
- To prevent this, a process needs a way to lock a resource to hold its assumptions

Multitasking Coding Practices

- Dangerous
  - Multiple calls access the same variable/resource
    - Globals, 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

Kernel

- The “core” OS functions are as follows
  - Perform scheduling – Handled by the scheduler
  - Dispatch of processes – Handled by the dispatcher
  - Facilitate inter-process communication

- A kernel is the smallest portion of OS providing these functions

Functions of an 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

Functions of an OS (continued)

- 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
    - Task Control Block (TCB) – a structure containing a task or a process
- Stored in a “Job Queue” implemented with pointers (array or linked list)

```
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
}
```

Task Control Block

- A TCB needs to be generic
  - A task can be just about anything the computer can do, a generic template can be used to handle any task
- Each task is written as a function conforming to a generic interface
  - Void aTask(void * taskDataPtr){
    - //task 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;
  - }

Kernel Example

- Tasks to be performed for this example:
  - Bring in some data
  - Perform computation on the data
  - Display the data
- First Implementation:
  - System will run forever cycling through each task calling the task and letting it finish before moving on
- Second Implementation
  - 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
  - Adds usage of ISR to avoid waiting

```c
// Building a simple OS kernel -step 1 #include <stdio.h>
// 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;
}
```

```c
// Building a simple OS kernel -step 2 #include <stdio.h>
// 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;
}
```

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:
  - task: display prompt
  - task: check if user entered data and move on otherwise .... implemented using interrupts
- Need ISR
  - How would you implement this with ISR?