Real Time Kernels and Operating Systems

Introduction
Typical embedded system solves a complex problem
By decomposing it into a number of smaller simpler pieces
That work together in an organized way
Such pieces called tasks
With multiple tasks system called a multitasking system

Several important responsibilities of a multitasking design include
Exchanging / sharing data between tasks
Synchronizing tasks
Scheduling task their execution
Sharing resources amongst the tasks

Operating System
Piece of software that provides the required coordination
When the control must ensure that task execution
Satisfies a set of specified time constraints
Called a real-time operating system

Primary software modules within an operating system
That implement such control
✓ Scheduler
✓ Dispatcher
✓ Intertask communication

Scheduler
Determines
Which task will run
When task will run

Dispatcher
Performs necessary operations
To start task

Intertask Communication
Mechanism for exchanging
Data and information between
Tasks or processes
Same machine
Different machines

Kernel
Smallest portion of operating system
Provides above services
Full Featured Operating System
  Provides additional libraries of functions
  Device drivers
  Rich communication packages
  Human computer interface

Real Time Operating System - RTOS
  Real time operating system is a special purpose operating system
  Implies rigid time requirements must be met
    If requirements not met results system functionality
    Inaccurate
    Compromised

Such systems usually interacting with physical environment
  Sensors
  Measurement devices

Usually in scientific experiments or control systems
Often people misuse term real time to mean system responds quickly

Come in two flavors
  Hard Real Time
    System delays are known or at least bounded
    Said to be operating correctly if can return results
    Within any time constraints

  Soft Real Time
    Critical tasks get priority over other tasks
    Retain priority until complete

    Real time task cannot be kept waiting indefinitely
    Tasks with time constraints
    Makes it amenable to mixing with other kinds of systems

Programs and Processes
  With the quick introductory overview of OS features and responsibilities
  Move inside
    Examine programs and processes
    What is a program…what is a process
    Let’s see

We have all worked with software programs

Program is not a process
  Program is passive entity
  Process is an active entity
A process is a program in execution
Execution proceeds in sequential manner
Single instruction at a time

Process includes more than just program code
Additionally includes
  State or program counter
    Which instruction is being executed
  Contents of program’s registers
    Working variables
  Process stack
  Temporary data
    Auto variables
    Subroutine parameters
    Return addresses
  Data section
    Reference to global variables
  Code section
    Reference to program instructions

Actually most of these are soft copies
  Of what may be held in pieces of hardware and used
    While process executing

Program may have several processes associated
  Each is considered to be a separate execution sequence

An embedded program is similarly a static entity
  Made up of a collection of firmware modules
  Can do no useful work unless it is running or executing
    Unless there are processes

When a firmware module is executing
  Again called a process or task

When a process is created
  It is allocated a number of resources by the system
    Can include
      ✓ Process stack
      ✓ Memory address space
      ✓ Registers (through the CPU)
      ✓ Program counter
      ✓ I/O ports
      ✓ Network connections
      ✓ File descriptors, etc
Resources generally not shared with other processes

During execution
  Contents of the program counter are continually changing
    As the process moves from instruction to instruction
      Working with data
  Currently executing instruction and present values of associated
    Collectively known as the process state
      Process state may contain
        Values of large number of other pieces of information
        As noted already

Resources and the CPU as a Resource
  Traditional view of computing focuses on the program not the processes
    One says that the program, running on the computer
      More specifically a task within program
        Set of processes comprising program

With embedded application
  Change the point of view from firmware to that of the microprocessor

Viewed with respect to the microprocessor
  More specifically the CPU
  CPU is simply another resource
    Available for use by the task
      To do its job

When a task enters the system
  Takes up space – memory
  Uses other system resources

Time that it takes to complete
  Called its execution time

Its persistence is duration
  From the time when it enters the system
  Until it terminates

Single Task
  If there only single process or task in system
    No contention for resources
    No restrictions on how long it can run
    How much memory it uses
Second Task

If a second process or task is added to the system
Potential resource contention problems arise
Generally only one CPU and the remaining resources limited

Problem resolved by
Carefully managing how the resources are allocated to each task
Controlling how long each can retain the resources
If each task shares the system’s resources
Each can get its job finished

If the CPU is passed between the tasks quickly enough
Will appear as if both tasks using it at same time

Will have system that models parallel operations
By time sharing a single processor

Certainly, the execution time for the program will be extended
However operation will give appearance of simultaneous execution

Such a scheme is called multitasking
Tasks said to be running concurrently.

Multiple Tasks
Concept can easily be extended to more than two processes or tasks

Implementing Control – A First Look
Under such a scheme
CPU is most important resource

In addition to the CPU however
Processes or tasks are sharing other system resources as well
Timers
I/O facilities
Busses

Despite the illusion that all of the tasks are running simultaneously
In reality at any instant in time
Only one process is actively executing
That process said to be in run state
Other process(es) in the ready waiting state
Such behavior is illustrated in the state and sequence diagrams.
For system with three tasks:
- One task will be running
- Others are waiting to be given the CPU

With ability to share CPU among several tasks:
Problem
- Deciding which task will be given the CPU
- When task will be given the CPU

Solution
Schedule is set up to specify:
- When
- Under what conditions
- For how long each task will be given use of CPU

Other resources

Criteria for deciding / controlling which task is to run next:
Collectively called a scheduling strategy

For embedded system such strategies generally fall into three categories:

* Multiprogramming
  - Running task continues until it performs an operation that requires waiting for an external event
e.g. waiting for an I/O event or timer to expire

* Real-Time
  - Tasks with specified temporal deadlines
  - Guaranteed to complete before those deadlines expire
  - Systems using such a scheme
  - Require a response to certain events
  - Within a well defined and constrained time

* Time-sharing
  - Running task is required to give up the CPU
  - So that another task may get a turn
  - Under a time-shared strategy
  - Hardware timer used to preempt the currently executing task
  - Return control to the operating system

Such a scheme permits one to reliably ensure:
- Each process is given slice of time to use operating system
Process State – Changing Context

Observed earlier: process is a program in execution
  Program in execution
  Comprises active processes
  As process executes it’s often changing state

Specifically at any time may be in any one of following states
- New
  Just being created
  - Running
    Instructions being executed
- Waiting
  Waiting for some event to occur
  I/O fetch for example
- Ready
  Waiting to be assigned to processor
- Terminated
  Finished execution

Task’s context comprises
  Important information about the state of the task
  Values of any variables
    Held in the CPU’s registers
  Value of the program counter
  State of the stack
  Etc.

Each time
  Running task is stopped – preempted or blocked
  CPU is given to another task
  That is ready
  Switch to a new context is executed

Context switch first requires
  State of the currently active task be saved

If task scheduled to get CPU had been running previously
  Its state is restored
  Continues where it had left off
Otherwise the new task starts from its initial state

As is evident context change
  Entails a lot of work
  Can take a significant amount of time
Earlier state diagram is now extended to reflect
  ✓ Task entering the system
  ✓ Being preempted
  ✓ Terminating

In multiprogrammed system
  Objective to have some processes running at all times
  Such scheme
    Maximizes CPU usage

As process Enters system
  Put into job queue
All jobs in system
  Contained in job queue
    User
    System
  Ready and Waiting processes in main memory
    Placed in ready queue
    Generally implemented as linked list
      Task or Process Control Blocks
      Each TCB (PCB) pointer field points to next job in queue

Other queues may exist in system as well
  Processes waiting for particular resource
    Placed in queue for that resource
    Often called device queue

New process initially put into ready queue
  Until selected for execution
  At such time
    Dispatched - given the CPU to execute

Dispatched process may have several events occur
  Issue I/O request and be placed in I/O queue
  Create new Child subprocess(es)
    Wait for their termination
  Could be returned to ready queue by CPU

In first two cases process enter Waiting state
  Eventually switch from Waiting to Ready state
  Returned to ready queue

Termination
  When process (is) Terminated
  Removed from all queues
  TCB and resources
    Deallocated
Task – Process Control Block
In task based approach
Each process represented in the operating system
By a data structure called Task Control Block – TCB
also known as a process control block

TCB contains all of the important information about the task
✓ A typical TCB contains following information
✓ Pointer (for linking the TCB to various queues)
✓ Process ID and state
✓ Program counter
✓ CPU registers
✓ Scheduling information (priorities and pointers to scheduling queues)
✓ Memory management information (tag tables and cache information)
✓ Scheduling information (time limits or time and resources used)
✓ I/O status information (resources allocated or open files)

TCB allocation may be static or dynamic
• Static allocation
  Typically used in embedded systems with no memory management
  Are a fixed number of task control blocks
  Memory is allocated at system generation time
  Placed in dormant or unused state.

When a task initiated
  TCB created
  Appropriate information entered

TCB is then placed into ready state by scheduler

From the ready state
  Will be moved to the execute state by dispatcher

When a task terminates
  Associated TCB returned to dormant state

With fixed number of TCBs
  No runtime memory management is necessary
  One must be cautious not to exhaust supply of TCBs

• With dynamic allocation
  Variable number of task control blocks
  Allocated from the heap at runtime
When a task initiated
   TCB created
   Appropriate information entered

TCB is then placed into *ready* state by scheduler

From the ready state
   Will be moved to the *execute* state by dispatcher

When a task terminates
   Associated TCB memory is returned to heap storage
   With a dynamic allocation
   Heap management must be supported

Dynamic allocation suggests an unlimited supply of TCBs
   However the typical embedded application has limited memory
   Allocating too many TCBs can exhaust the supply

Dynamic memory allocation scheme
   Generally too expensive for smaller embedded systems

Queues
When a task enters the system
   Typically be placed into a queue called the *Entry Queue* or *Job Queue*

Easiest and most flexible way to implement such a queue
   Utilize a linked list as the underlying data structure
   Last entries in the TCB
      Hold the pointers to the preceding and succeeding TCBs

Whether queue, an array, or some other data type used to hold TCBs
   Entries must all look alike
      Such a requirement will impose some restrictions on implementation

In C
   TCB is implemented as a struct
      Containing pointers to all relevant information
      Because the data members of a struct must all be of the same type
         Pointers are all void* pointers.
   Skeletal structure for a typical TCB identifying essential elements
      Task
      Example set of task data
   Given in the following C declarations
Threads – Lightweight and Heavyweight

Task or process characterized by
Collection of resources utilized to execute program

Thread
Smallest subset of these resources necessary for the execution of the program
Copy of the CPU registers
Including the program counter and a stack

Sometimes the subset of resources
Called a lightweight thread

In contrast to the process itself
Referred to as a heavyweight thread

Thread can be in only one process
Process without a thread can do nothing

```c
#include <stdio.h>

// The task control block
struct TCB
{
    void (*taskPtr)(void* taskDataPtr);
    void* taskDataPtr;
    void* stackPtr;
    unsigned short priority;
    struct TCB* nextPtr;
    struct TCB* prevPtr
};

struct TCB* TCBList;

// The data passed into the task
struct taskData
{
    int taskData0;
    int taskData1;
    char taskData2;
    // The task
    void aTask(void* taskDataPtr)
    {
        function body;
    }
```

```c
```
Single Thread
Sequential execution of a set of instructions
Through a task or process in an embedded application
Called a thread of execution, or thread of control

Thread
Has a stack and status information relevant to its state and operation
*Copy* of the (contents of) the physical registers

During execution uses
  Code (firmware)
  Data
  CPU (and associated *physical* registers)
  Other resources allocated to the process

Diagram presents
Single task with one thread of execution

Model is referred to as a *single process – single thread design*.

When we state that process is *running*, *blocked*, *ready*, or *terminated*
Are actually describing different states of thread

If embedded design intended to perform a wide variety of operations
  With minimal interaction
  May be appropriate to allocate one process
    To each major function to be performed

Such systems ideal for *multi process – single thread* implementation

Multiple Threads
Many embedded systems intended to perform single primary function
Operations performed by function all interrelated

During partitioning and functional decomposition
  Seek to identify which actions benefit from parallel execution
    Might consider allocating subtask for each type of I/O

Nature of application executing as a single primary function
Suggests that associated process should be decomposed
  Into a number of subtasks
    Executing in parallel

At runtime process can pass the CPU around to each of subtasks
  Thereby enabling each to do its job
Each of the smaller jobs has its own thread of execution
Such a system called a *single process – multithread design*

Unlike processes or tasks
- Threads are not independent of each other
- Can access any address within the process
  - Including other thread’s stacks
Why is this important to note?

Context switch between threads
- Can be substantially simpler and faster
  - Than between processes

When switching between threads
- Much less information must be saved and restored

An operating system that supports tasks with multiple threads
- Referred to as a *multithreaded operating system*

Can easily extend design to support multiple processes
- Can further decompose each process into multiple subtasks
- Such a system called *multiprocess – multithread design*

**Sharing Resources**

Based upon discussions
- Can identify four categories of multitasking operating system
  - ✓ *Single process–single thread*
    - Has only one process
      - In embedded application that process runs forever
  
  - ✓ *A multi process–single thread*
    - Supports multiple simultaneously executing processes
      - Each process has only single thread of control

  
  - ✓ *A single process–multiple threads*
    - Supports only one process
      - Within the process has multiple threads of control

  
  - ✓ *A multi processes – multiple threads*
    - Supports multiple processes
      - Within each process is support for multiple threads of control

Major distinguishing feature
- Which resources process and hence thread(s) is / are using
- Where the resources come from
At a minimum process or task will need

✓ Code or firmware – the instructions.
  Are in memory and have addresses.
✓ Data that the code is manipulating
  Starts out in memory
  May be moved to registers
  Data has addresses
✓ CPU and associated physical registers
✓ Stack
✓ Status information

First three items
Shared among member threads
Last two are proprietary to each thread

Each thread has *copy* of the registers
Often other necessary resources
  Timers
  Measurement
  Signal generation resources
  I/O ports etc.

**Memory Resource Management**

**System Level Management**

After CPU
Memory probably most important resource available to task
Spend some time to examine characteristics
Unique to embedded systems

Most microprocessor designs today still based upon von Neumann architecture
Program (instructions) stored memory
In same manner as any other piece of information (data)
With single physical memory
Instructions and data accesses
Use same physical bus
Limits execution speed

When process created by the operating system
Is given portion of physical memory in which to work
Set of addresses (a resource) delimiting that code and data memory
Proprietary to each process called its *address space*
Address space typically not shared
With any other peer processes

When multiple processes concurrently executing in memory
Errant pointer or stack error can easily lead to
Memory owned by other processes
Being inadvertently accessed
Overwritten
System software must restrict the range of addresses
Accessible to the executing process
Process (thread) trying to access memory
Outside its allowed range

Should be immediately stopped
Before it can inflict damage on memory belonging to other processes

One means by which such restrictions are enforced
Concept of privilege level

Processes are segregated into
User mode capability
User mode limits the subset of instructions process can use
Supervisor mode capability
Can access entire memory space

Processes with low (user mode) privilege level
Not allowed to perform certain kinds of memory accesses
Not allowed to execute certain instructions

When a process attempts to execute such restricted instructions
An interrupt is generated
Supervisory program with a higher privilege level
Decides how to respond

Supervisor mode privilege level
Generally reserved for supervisory or administration types of tasks
Delegated to the operating system
Processes with such privilege
Have access to any firmware
Can use any instructions within the microprocessor’s instruction set
Process Level Management

Process may create or spawn *child processes*

Parent process may choose to give a subset of its resources
To each of the children

Children are separate processes
Each has its own
- Data address space
- Data
- Status
- Stack

Code portion of address space shared

Process may create *multiple threads*

Parent process shares most of its resources
With each of the threads

Are not separate processes but separate threads of execution
Within the same process
Each thread will have
- Its own stack and status information

In contrast to lightweight threads
*Processes or tasks* exist in separate address spaces
One must use some form of messaging or shared variable
For inter-task exchange

Processes have stronger notion of encapsulation than threads
Each *thread*
- Has own CPU state
- Shares with peer threads
  - Code section
  - Data section
  - Task resources
- Sharing gives threads weaker notion of encapsulation

Re-entrant Code

Child processes and their threads
- Share same firmware memory area
As a result two different threads
- Can be executing the same function at the same time

Functions using *only* local variables
- Inherently *re-entrant*
  - They can be simultaneously
    - Called and executed in two or more contexts
Local variables
Copied to stack
Each invocation will get new copies

Functions that use
Global variables
Variables local to the process
Variables passed by reference
Shared resources
Not re-entrant

One must ensure all accesses to any common resources are coordinated

When designing the application…must make certain
One thread cannot corrupt the values of the variables in a second
Any shared functions must be designed to be re-entrant

Design said to be *thread safe*
If code functions correctly
During simultaneously execution
By multiple threads
In same address space

**Controlling the System**
In world of embedded systems
Can control operation of systems in number of different ways
Can loosely classify such systems into two broad categories
Time based
Reactive

Let’s look at each

**Time Based Systems**
Systems whose behaviour controlled by time
Can be
Absolute
Relative
Following an interval

✓ Absolute time
Real world time

✓ Duration
Relative time measure
Non-equal intervals
Interval
Distinct from duration
Interval marked by
Specific start and end times
Equal intervals have same start and stop
Can have same duration

Operating Systems
Operating system
Special and powerful subclass of time based systems
Types commonly found in embedded applications
Full operating system
Subset of full system
RTOS - Real time operating system
Special class of OS
With constraints on system level timing
Embedded operating system provides an environment
Within which firmware pieces – tasks are executed

Easiest way to first view an operating system
From the perspective of the services it can provide

Internally operating systems vary greatly
In both design and the strategy for delivering such services

Operating system must provide or support four specific functions.
- Schedule task execution
- Dispatch a task to run
- Ensure communication and synchronization amongst tasks
- Manage resources

Scheduler
Determines
Which task will run
When it will do so

Dispatcher
Performs the necessary operations to start task

Intertask or interprocess communication
Mechanism for exchanging data and information
Between tasks or processes
On the same machine
On different one

Kernel is the smallest portion of operating system
That provides these functions
Easiest way to view is from perspective of services provided

Include

**Process Management**
- Creation and deletion of user and system processes
- Suspension and resumption of processes
- Manage interprocess communication
- Handle and resolve deadlocks

**Main Memory Management**
- Track which parts of memory are being used
- Track which processes are loaded into memory
- Allocate and deallocate memory space as needed

**Secondary Memory Management**
- Manage free disk space
- Storage allocation
- Disk scheduling

**I/O System Management**
- General device driver interface
- Caching and buffering of I/O
- Device drivers for specific devices

**File System Management**
- Creation and deletion of files
- Directory creation, deletion, and management
- Mapping onto secondary storage
- Backup onto nv storage

**System Protection**
- Managing concurrent users and processes
- Ensuring protection of data and resources
Networking
Manages intrasystem communication and scheduling of tasks

Command Interpretation
Provides the interface between the user and the operating system

Layering and Virtual Machines
Most contemporary operating systems
Implemented using a layered approach
Main advantage is increased modularity
Layers are designed such that each layer
Uses functions / operations and services of lower layers
Typical architecture appears as
Shown on left
In some layered implementations
Such as that on right
Higher level layers have access to lower level
System calls
Hardware instructions
With such capability
Can make application programmers interface
Appear to be machine itself
Can logically extend concept
Using scheduling and virtual memory concepts
Can create illusion
Each program running on its own machine
 Called virtual machine concept
With such a machine
No reason why could not run entirely different
Operating system
DOS on UNIX
Associated software packages

Virtual machine can be difficult to implement in general
Not always a good match between
Hardware
3 disk drives for example
Virtual machines
More than 3
Each can’t have it’s own drive
Must create virtual disks
Other more complex system level issues
Reactive or Foreground – Background Systems

*Reactive systems*
Comprise tasks
  Initiated by some event
  Internal or external to system

  **Internal event**
  Internal timer interrupt
  May be elapsed time
  Bound on data exceeded

  **External event**
  External world interrupt
  Recognition of keystroke or switch activated
  External response to internally generated command

Such systems do nothing until event occurs
Called *event driven systems*
The *foreground / background model* for managing task execution
Decomposes set of tasks into two subsets
  Called *background tasks* and *foreground tasks*

Traditional decomposition

*Foreground set*
Tasks that interact with the user or other I/O devices

*Background set*
Remainder

Interpretation is slightly modified in the embedded world

- Foreground tasks
  Those initiated by interrupt or by a real-time constraint that must be met
  They will be assigned the higher priority levels in the system

- Background tasks
  Non-interrupt driven and are assigned the lower priorities
  Once started will typically run to completion
    Can be interrupted or preempted by any foreground task at any time

  Should include those that do not have tight time constraints
  Good candidates include tasks designed to
    Continuously monitor system integrity
    That involve heavy processing are

Often separate ready queues will be maintained for the two types of tasks
Representing Time

When considering either time-based or reactive systems, time is an important element.

- **Time based**
  - When does something occur?
  - How tightly can time intervals be held or met?

- **Reactive**
  - How quickly can an event be recognized?
  - How quickly and repeatedly can an event be responded to?

Issues of time are important when trying to schedule tasks and threads. Tasks or threads that are initiated with repeating duration between invocations are called **periodic**. Such duration is called the **period**. Time to complete a called execution is called **execution time**. Variation in evoking an event is called **jitter**. We must examine each context to determine the significance of jitter with respect to time constraints.

Let’s see how we can express this. Consider basic heart pace maker.

Figure illustrates Edmark wave of heartbeat.

Normal operation:
- Ventricular sense: Hearting filling with oxygenated blood
- Pump: Heart muscle contracts to pump blood
- Refractory period: Muscle fiber relaxes until can fill and contract.

We can express changes in state in our systems in variety of different ways:
- Timing diagram
- State chart
Activity diagram

Simplest method probably a timing diagram
Timing diagram in this context
Different from what may have encountered
Here we express behaviour of system
Moving between states
In basic diagram we express
States along vertical axis
Time along horizontal axis

We elaborate by annotating
Durations
Events
Jitter
State transitions
Here we illustrate a periodic system

The sloped lines indicate transition between states
Such transition potentially may be significant
Most of time small compared to other times
Observe how leading and trailing jitter represented

Can show same thing for aperiodic sequence

Note we specify min and max times

Invocation of aperiodic tasks varies
Duration between such tasks called \textit{inter-arrival time}
Such time is critical when determining how to schedule
Real time tasks

Under such circumstances
Must identify lower bound on inter-arrival time
May also need to consider such things as
Maximum number of events within given time interval

\textbf{Thinking Schedule – A First Look}

When working with scheduling system
Must address \textit{priority} of task
Priority based upon different criteria
Will examine these shortly

Used to resolve which task to execute
When more than one task
Waiting and ready to execute
Tasks with higher priority
   Execute preferentially over those with lower priority

Real time system one in which correctness implies timeliness
Most such systems carefully manage resources
   To ensure maintaining predictability
      Of the timeliness constraints
Predictability gives measure of accuracy
   With which one can state in advance
      When and how an action will occur
Thus schedule a real-time system

Task which must start or finish by specified time
   Defined as hard or said to have a hard deadline
Missed deadline considered to be
   Partial or total failure
✓ System is defined as hard real-time
      If contains one or more such tasks
         Such system may have other not or non hard real-time tasks
         Major focus however on hard deadlines

✓ System with relaxed constraints defined as soft real-time
   Such systems may meet deadline on average
   Soft real-time systems may be soft in several ways
      • Relaxation of constraint that missing deadline
         Constitutes system failure
         Such system may tolerate missing specific deadline
            Provided some other deadline or timeliness constraint met
            Average throughput for example
      • May evaluate correctness of timeliness as
         Gradation of values rather that pass or fail
         How bad did we miss deadline

✓ System with tasks having some constraints (but relaxed) as well as hard deadline
   Defined as firm real-time

Task that can be determined to always meet timeliness constraint
   Said to be schedulable
Task that can be guaranteed to always meet all deadlines
   Said to be deterministically schedulable
   Occurs when event’s worst case response time
      Less than or equal to task’s deadline

When all tasks can be scheduled
   Overall system can be scheduled
Does it matter
Following table captures timeliness constraints
With respect to whether task is soft or hard real-time

<table>
<thead>
<tr>
<th>Property</th>
<th>Non Real-time</th>
<th>Soft Real-time</th>
<th>Hard Real-time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deterministic</td>
<td>No</td>
<td>Possibly</td>
<td>Yes</td>
</tr>
<tr>
<td>Predictable</td>
<td>No</td>
<td>Possibly</td>
<td>Yes</td>
</tr>
<tr>
<td>Consequences of late computation</td>
<td>No effect</td>
<td>Degraded Performance</td>
<td>Failure</td>
</tr>
<tr>
<td>Critical reliability</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Response dictated by external Events</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Timing analysis possible</td>
<td>No</td>
<td>Analytic (sometimes), stochastic simulation</td>
<td>Analytic, stochastic simulation</td>
</tr>
</tbody>
</table>

Scheduling Tasks
With all this information in hand
Let’s get to work and examine process of scheduling tasks

Scheduling comes in during design phase of our development
We decide the schedule

Involves decisions that affect and optimize
Overall performance of our system
According to some criteria
Given in the specification

When dealing with hard deadlines
Must ensure that such tasks and associated actions
Can meet every deadline

Soft deadlines
Give us more flexibility
Now focus is on trying to minimize items such as
Missed deadlines
Delay in initiating task

Success of CPU scheduling depends upon following
Observed property of processes
Process execution consists of cycle of
CPU execution
I/O wait
CPU and I/O bursts alternate until process completes

Frequency of bursts tends to be fairly predictable independent of
Machine or process
Generally characterized as exponential
Given in figure
Whenever CPU becomes idle
Which it may do during I/O times
Operating system must select process
From ready queue
To be executed
Process carried out by scheduler
According to predefined algorithm

CPU scheduling is basis of multi-programmed operating
By switching CPU among processes
Operating system makes computer more productive
Recall we refer to this as context switch
For real time systems
Time required for context switch critical

As noted our goal is to have some process running at all times
Whenever CPU becomes idle
Operating systems selects process in ready queue to execute
Note ready queue not necessarily a FIFO
Next (ready) job selected may be based upon other criteria
Priority
Basic queue with priority associated with each entry
Get from the head
Search the queue to find job that meets priority criteria
Return it

Scheduling Decisions
Two key elements of real-time design
Repeatability
Predictability
These are absolutely essential in context of hard deadlines

To ensure predictability
We must
Completely define the timing characteristics of tasks
Properly schedule using predictable scheduling algorithm
Not always easy as we’ll see shortly

When do we select a new task to run

Preemptive vs Non-Preemptive Scheduling
Decisions made under following four conditions
1. Process switches from running to waiting states
I/O request
2. Process switches from running to ready state
   When interrupt occurs
3. Process switches from waiting to ready
   Completion of I/O
4. Process terminates

If only using conditions 1 and 4
   New process must be scheduled
   Such scheduling called non-preemptive
Under such scheduling
   Process keeps CPU until it releases
   Terminating
   Switching to waiting state
Otherwise scheduling called preemptive

Schedulers in real-time systems
   Assign priority to each task
   As noted earlier establishes precedence of task
   When multiple tasks ready to run
Most common scheduling policy in such systems
   Use preemptive scheduling
If lower priority task executing
   Arriving higher priority task preempts
   Lower priority task suspended
   Resumes when higher priority task completes
Otherwise
   Runs to completion

Although priority scheme seems to ensure
   Higher priority tasks will always complete
Not always the case
With preemption problem of blocking arises
Blocking occurs when task needs resource
   Owned by another task

Consider several examples
Case 1:
   Task A has higher priority than task B
   Task B starts and reserves resource R1
   Task A preempts Task B
   Task A begins execution and becomes blocked at point
   Resource R1 needed
   Task A must suspend and allow B to complete
   Thereby releasing R1

Second case introduces problem called priority inversion
Problem of this nature occurred on one of Mars missions

Case 2:
We have 3 tasks
  Task A, Task B, and Task C
  Task A has highest priority and Task C the lowest
Task C starts and reserves resource R1
Task A preempts Task C
Task A begins execution and becomes blocked at point
  Resource R1 needed
Task A must suspend and allow C to continue
  Hopefully releasing R1
Task B preempts Task C and does not need R1
  Task B completes
  Allows Task C to resume
Easy to create situation in which highest priority task
  Blocked forever
See that high priority task
  That can be scheduled in isolation
  May fail in multitasking context

In hard real-time context
  Must ensure bound on priority inversion

Additional Scheduling Criteria
Must ask what is important
  Number of different scheduling algorithms

In making choice must consider properties of various algorithms

Other properties include
  • CPU utilization
    Want to keep as busy as possible
    Ideally 100 per cent
In real system should range between
  40% for lightly loaded system
  90% for heavily loaded system
Also speak of utilization with respect to single task
For such a periodic task, Ti, utilization given as
  \[ u_i = \frac{e_i}{p_i} \]
    \( u_i \) fraction of time task keeps CPU busy
    \( e_i \) execution time
    \( p_i \) for periodic task is the period
Can express similar relationship for aperiodic tasks
- Throughput
  Number of processes that are completed per unit of time
  Depends on course on complexity of process / task

- Turnaround Time
  Interval from time of submission of task until its completion
  Includes time
    - Waiting to get into memory
    - Waiting in ready queue
    - Executing on CPU
    - Doing I/O

- Waiting Time
  Scheduling algorithm execution and I/O time
  Affects only time spent in waiting queue
  Includes all time in waiting queue

- Response Time
  For interactive system
  - Turnaround time may not be best measure
  - Consider time from submission to first response
  - Time take to first response not time to first output

Scheduling Algorithms
  Beyond scope to go into details of all scheduling algorithms
  Let’s look at several however
  - We’ll begin with very simplest

Polled and Polled with Timing Event
  Simple kernel designed for use with single task
  - Although simple
    - Algorithm sometimes essential in cases
    - With hard deadline
  Important to recognize the significance of time

Polled
  Among simplest and fastest
  System continually loops
  - Looking for event to occur
  Works well for single task
  - Deterministic
  - Time to respond to event
    - Computable
    - Bounded
    - Worst case
Assume event occurs immediately after test instruction
Response time is length of loop

**Polled with Timing Event**
- Simple extension
- Uses timing element
  - Delay action after polled event true
- Can be used to deskew signals

**Timing Interrupt / Event Interrupt Driven**
- System continually loops
  - Until interrupted by
    - Timing event
      - Typically internal signal
    - Interrupting event
      - Typically external signal
- Uses timing/interrupt event to trigger context switch
- **Hardware**
- **Software**
- **Timing**
  - **Periodic**
    - Fixed rate scheduling
  - **Aperiodic**
    - Sporadic scheduling

Can work with multiple tasks
  - Basis for time-sharing systems

Tasks may or may not be equal
  - **Periodic**
    - All given same amount of time
  - **Aperiodic**
    - Time allocation based upon priority

**First-Come-First-Served**
- Simple algorithm is first-come first-served
  - Easily managed with FIFO queue
  - When process enters ready queue
    - TCB linked to tail of queue
  - When CPU free
    - Allocated to process at head of the queue
    - Running process removed from queue
  - Is non-preemptive algorithm
    - Can be troublesome in real-time system
Shortest Job First
Assumes CPU used in bursts of activity
Each task has associated estimate of
How much time job will need when next given CPU
Estimate is based upon measured lengths of previous CPU usage
Can be either preemptive or non-preemptive
With preemptive schedule
Currently running process can be interrupted by one with
Shorter remaining completion time

Priority Schedule
Shortest job first
Special case of more general priority scheduling
Priority associated with each process
CPU allocated to process with highest priority
Equal priority jobs scheduled first-come first-serve
Major problem
As we’ve discussed have potential for
Indefinite blocking or starving
Priority inversion
Can be either preemptive or non-preemptive
Can make priority decisions
During design
Static schedule
During runtime
Dynamic schedule

• Rate Monotonic
With preemptive schedule
Currently running process is interrupted by one with
Higher priority
Special class called rate-monotonic
Initially developed in 1973
Updated over the years
In basic algorithm priority assigned based upon
Execution period
Shorter period – higher priority
Priorities determined and assigned at design time
Remain fixed during execution
Said to use static or fixed scheduling policy
We compute schedulability as bound on utilization of CPU

Sum on left hand side

Individual task utilizations
For \( n = 1 \)
Have 100% utilization
As \( n \to \infty \)
Utilization \( \to 69\% \)

e and \( p \)
Execution time and period of task respectively

\[
\sum_{i=0}^{n-1} \frac{e_i}{p_i} \leq n \left( \frac{1}{2^n} - 1 \right)
\]

Approach makes following assumptions
Deadline for each task
Equal to its period
All tasks preemptible at any time

The expression on right hand side
Gives bound on utilization
Establishes extreme bound
If cannot be met
Must execute more detailed analysis
To prove schedulability
Sets bound at 69% utilization
Practically could be relaxed to 88%
Still be scheduled

Basic algorithm given above
Simplifies system analysis
Scheduling is static
Worst case occurs when all jobs started simultaneously

Rate monotonic schedule – \textit{critical zone theorem}

If the computed utilization is less than the utilization bound, then the system is guaranteed to meet all task deadlines in all task phasings.

Can be shown rate-monotonic systems are
Optimal fixed rate scheduling method
If rate-monotonic schedule cannot be found
No other fixed rate scheme will work
Stable

Note: priority is based upon execution period

As additional lower priority tasks added
Higher priority tasks can still meet deadline
Even if lower priority tasks fail to do so
Assumes no blocking

Basic algorithm can be modified to include blocking

\[
\sum_{i=0}^{n-1} \frac{e_i}{p_i} + \max \left( \frac{b_0}{p_0}, \ldots, \frac{b_{n-1}}{p_{n-1}} \right) \leq \frac{1}{2^n - 1}
\]

Terms \( b_i \) give maximum time task \( i \) can be blocked
By lower priority task

With non-preemptive schedule
Currently arriving higher priority process
Placed at head of ready queue

- **Earliest Deadline**

Earliest deadline uses a dynamic algorithm
Priority assigned based upon task with closest deadline
Must be done during runtime
Only then can deadline(s) be assessed
Set of tasks considered schedulable
Sum of task loading less than 100%

Considered optimal
Sense if can be scheduled by other algorithms
Can be scheduled by Earliest Deadline
Algorithm not considered stable
If runtime task load rises above 100%
Some task misses deadline
Generally not possible to predict which task will fail

Further adds runtime complexity
Scheduler must continually determine
Which task to execute next
Whenever such decisions must be made
Analytical methods more complex than fixed priority cases
• **Least Laxity**
  This algorithm similar to earliest deadline
  Constraint a little tighter
  In addition to deadline
  Considers time to execute task
  Which task has least room to move

  Thus
  Priority based upon
  laxity = deadline – execution time
  Task with negative laxity
  Cannot meet deadline

  Schedule based upon ascending laxity
  On paper rather straight forward concept
  However means
  Must know
  Exact value or upper bound on execution time
  Must update values
  With each system change

  Can utilize in system with hard and soft deadlines
  Hard time tasks can be given priority
  Over those with less rigid constraints

  Has weaknesses similar to Earliest Deadline
  Not stable
  Greater run time burden than fixed schemes
  Tends to devote CPU cycles to tasks
  That are clearly going to be late
  Causes more tasks to miss deadlines

• **Maximum-Urgency-First**
  Algorithm includes features of
  Rate Monotonic
  Least Laxity

  First cut
  Assign priority according to period
  Same as Rate Monotonic
Add binary *criticality* task parameter
   Parameter does decomposition into two sets
      Critical and non-critical
   Least Laxity algorithm
      Applied to those in critical set
      Observe this is done at runtime

If no critical tasks waiting
   Tasks from non-critical set scheduled
Because critical set based upon Rate Monotonic algorithm
   Can structure so that no critical task
      Fails to meet deadline

Major advantage of algorithm
   Simplicity of static priority
   Reduced runtime burden compared with full Least Laxity

Lacks some flexibility
   Rate monotonic assumes unconstrained preemption
      Short deviations typically tolerated well
   Longer deviations
      Can lead to missed deadlines
Best applied
   Tasks well understood
   Blocking constraints easy to determine
   Dynamic scheduling contribution from Least Laxity
      Potentially can compensate by elevating task’s priority
   Has some of runtime complexity of pure Least Laxity

Round Robin
   Designed especially for timeshared systems
   Similar to first-come first-served
   Preemption added to switch between processes
   Small unit of time called *time quantum* or *slice* defined
Ready queue treated as circular queue
   Scheduler walks queue
      Allocating CPU to process for 1 time slice
      If process completes in less than allocated time
         It releases CPU
      Else the process is interrupted when time expires
         Put at end of queue
   New processes are added to tail of queue

Observe
   If time slice increased to infinity
      Becomes first-come first-served scheduler
Real Time Scheduling Considerations

Have noted real time system may be

- Hard real time
- Soft real time

Scheduling may be

- Static
- Dynamic

**Hard Real Time**

If dynamic

- General process submitted along with statement
  - Time required to compute and do I/O

Scheduler

- Accepts process
  - Guarantees can complete on time
- Rejects as impossible

Called *resource reservation*

- Requires scheduler to know exactly how long
  - Each operating system function takes
  - Requires completion time guarantee
  - Impossible for systems with
    - Secondary storage
    - Virtual memory

**Soft Real Time**

Less restrictive

- Require critical processes to have priority
  - Over less critical

Implementing soft real-time system

- Requires careful design of
  - Scheduler
  - Related aspects of operating system

Requires

- Priority scheduling
  - Real time processes must have highest priority
  - Must not degrade over time
  - Relatively easy to ensure
- Dispatch latency must be small
  - Requires system calls to be preemptible
  - Achieved several ways
    - Insert preemption points
    - Check if high priority process needs tp be run
  - Make entire kernel preemptible
  - All kernel data structures must be protected
  - Synchronization methods
Comprised of two components
Conflict phase
Preemption of any process running in kernel
Low priority process releasing needed resources
Context switch to high priority process
Dispatch phase
Moving from ready state to run state

Algorithm Evaluation
As we’ve seen
There are many algorithms each with own parameters
Selecting difficult
Must first establish criteria
CPU utilization
Response time
Throughput
Next must evaluate algorithms against criteria
Variety of methods - let’s examine several

Analytic Evaluation
Major class of methods called analytic evaluation
Uses algorithm and system workload
Produce formula or number to evaluate algorithm
For workload
One such method called deterministic modeling
Takes predetermined workload
Defines performance of each algorithm for workload
Consider following processes and workloads

<table>
<thead>
<tr>
<th>Process</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>10</td>
</tr>
<tr>
<td>P2</td>
<td>29</td>
</tr>
<tr>
<td>P3</td>
<td>3</td>
</tr>
<tr>
<td>P4</td>
<td>7</td>
</tr>
<tr>
<td>P5</td>
<td>12</td>
</tr>
</tbody>
</table>

Let’s look at the following scheduling algorithms
First Come First Served
Shortest Job First
Round Robin
FCFS

<table>
<thead>
<tr>
<th>Process</th>
<th>Waiting Times</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>P1 10</td>
</tr>
<tr>
<td></td>
<td>P3 3</td>
</tr>
</tbody>
</table>

Waiting Times
Average = 28 units

SJF

<table>
<thead>
<tr>
<th>Process</th>
<th>Waiting Times</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>P3 3</td>
</tr>
</tbody>
</table>

Waiting Times
Average = 13 units

RR

Preempt every 10 time units

<table>
<thead>
<tr>
<th>Process</th>
<th>Waiting Times</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>P1 10</td>
</tr>
<tr>
<td></td>
<td>P2 10</td>
</tr>
</tbody>
</table>

Waiting Times
Average = 23 units

Deterministic modeling
Simple and fast
Requires exact knowledge of process times
Often difficult to establish
One solution is to measure over repeated executions

Queuing Models
Processes run on many systems vary from day to day
No static set of processes and times
For use in deterministic modeling
Can measure or compute distribution of CPU and I/O bursts
Have seen this is typically exponential
Can be described by a mean value
Can determine similar distribution for process arrival times
Based upon two distributions
For most algorithms possible to compute average
Throughput
Utilization
Waiting times
etc.
Can model computer as collection or network of servers
Each server has queue associated
Knowing arrival and service rate
Can compute
Utilization
Average queue length - n
Average wait time - w
Let average arrival time be $\lambda$
Thus
If system in steady state
Number of processes leaving = number of process arriving

$$n = \lambda \times W$$
Known as Little’s formula
Useful because valid for any scheduling algorithm
Knowing any two variables
Can compute third

Useful for comparing algorithms
Has limitations
Mathematics of complex algorithms and distributions
Difficult to work with
Arrival and service distributions complex
Queuing models
Only approximation of real system

**Simulation**
To get more accurate evaluation of scheduling algorithm
Can use simulations
Requires
Models of computer system and processes
Data to drive simulation
Often collected from trace of actual processes
Recording of actual events
On real system
Can be expensive
Becoming increasingly powerful tool
Implementation

Build and test
Most accurate method
Difficulty is cost
   Development
   System to support