Operating Systems

Babak Kia
Adjunct Professor
Boston University
College of Engineering
Email: bkia -at- bu.edu

What is an Operating System

- **Operating System handles**
  - **Memory Addressing & Management**
  - **Interrupt & Exception Handling**
  - **Process & Task Management**
  - **File System**
  - **Timing**
  - **Process Scheduling & Synchronization**
- **Examples of Operating Systems**
  - **RTOS – Real-Time Operating System**
  - **Single-user, Single-task: example PalmOS**
  - **Single-user, Multi-task: MS Windows and MacOS**
  - **Multi-user, Multi-task: UNIX**

Real-Time Operating Systems

- **Operating systems come in two flavors, real-time versus non real-time**
- **The difference between the two is characterized by the consequences which result if functional correctness and timing parameters are not met in the case of real-time operating systems**
- **Real-time operating systems themselves have two varieties, *soft real-time* systems and *hard real-time* systems**
- **Examples of real-time systems:**
  - **Food processing**
  - **Engine Controls**
  - **Anti-lock breaking systems**

Soft versus Hard Real-Time

- In a ***soft real-time*** **system, tasks are completed as fast as possible without having to be completed within a specified timeframe**
- **In a** ***hard real-time*** **operating system however, not only must tasks be completed within the specified timeframe, but they must also be completed correctly**

4

Foreground/Background Systems

- **The simplest forms of a non real-time operating systems are comprised of** ***super-loops*** **and are called** ***foreground/background*** **systems**
- **Essentially this is an application consisting of an infinite loop which calls functions as may be necessary to perform various tasks**
- **The functions which are called to perform these tasks are** ***background functions*****, and are executed at the** ***task-level*** **of the operating system**

5

F/B Systems

- **On the other hand, processes which must be handled in a timely fashion such as interrupts are** ***foreground processes*** **and are executed at** ***interrupt-level***
- **Most microcontroller based embedded systems, such as microwaves and washing machines are foreground/background systems**

6

Definitions

- The *Critical Section* of a code is code which needs to be executed indivisibly and without interruption. The operating system usually disables interrupts before entering a critical section, and enables them again after its completion
- A *resource* is any object used by a task. It can be anything from an I/O pin to a data structure
- *Shared resources* are resources that can be used by more than one task. However, to prevent data corruption, each task needs to obtain exclusive access to the shared resource through a mechanism called *mutual exclusion*

7

Definitions

- A *task*, also referred to as a *thread*, is an independent section of a program complete with its own stack and CPU register space
- Each task is assigned a priority, and is always placed in one of *dormant, ready, running, waiting,* or *ISR* states
- A *dormant* task is a task which is available in memory but is not submitted to the kernel for execution

8

Definitions

- A task is deemed *ready* when it is available for execution but its priority is less than the current task priority of the system
- Consequently, a task is *running* when its current priority is met and the CPU starts to execute it
- A task is considered in *wait* mode when it is waiting for a resource to become available

9

Definitions

- And finally a task is considered *interrupted* when the CPU is in the process of servicing an interrupt
- If a task needs to be put on hold so that another task can execute, a context switch or a task switch occurs
- In this event, the processor saves the task's context (CPU registers) usually into a secondary storage area, and loads the new task's context from the same

10

Kernels

- The engine within the operating system which is in charge of handling tasks and communications between them is called the *kernel*
- A real-time kernel for instance manages breaking up an application into a series of tasks to be performed in a *multi-tasking* fashion
- Multitasking systems maximize the usage of a CPU and allow programmers to easily manage the complexities associated with real-time systems

11

Kernels

- Kernels come in two flavors, *pre-emptive* kernels, and *non pre-emptive* kernels
- *Pre-emptive kernels* are used when system response times are of critical concern, and as such most real-time operating systems are pre-emptive in nature
- Non pre-emptive kernels require that the tasks themselves give up using the CPU, and therefore this is a process which must be performed frequently

12

Non Pre-emptive Kernels

- One advantage of using a non-pre-emptive system is that interrupt latencies are low
- Another advantage is that programmers can use non re-entrant functions within their code
- This is because each task runs to completion before another task executes
- By the same token, there is less headache associated with the management of shared resources

13

Non Pre-emptive Kernels

- The biggest disadvantage of a non pre-emptive kernel is task responsiveness. This is because a high priority task is made to wait until the current task (even if of a lower priority) has finished execution
- Interrupts can preempt a task, however even if a higher priority task is scheduled within an interrupt service routine, it still cannot run until the CPU operation is relinquished by the current task

14

Pre-emptive Kernels

- As mentioned earlier, most real-time operating systems are pre-emptive
- This is because the execution of a higher priority task is deterministic
- However, programs written to run on a pre-emptive system must only use re-entrant functions in order to guarantee that both a low and a high priority task can use the same function without fear of data corruption

15

Pre-emptive Kernels

- One important difference between pre-emptive and non pre-emptive systems is that upon executing an interrupt service routine, the pre-emptive system always runs the highest ready task (not necessarily the task which was interrupted), whereas a non pre-emptive system returns to the task which was interrupted

16

Reentrancy

- Not just a dirty word!
- *Reentrant functions* are critical to the proper operation of a preemptive OS
- A reentrant function can be used by more than one task without fear of data corruption
- Also, a reentrant function can be interrupted at any point and restarted at a different time without loss of data
- Reentrancy is achieved by using local data, or by protecting global data

17

Priority

- Tasks need a mechanism by which they can be prioritized
- This is done at the kernel level and at compile time by the programmer
- Task priorities can be *static*, in that the priority of a task does not change for the duration of the application's execution
- On the other hand, priorities are deemed *dynamic* if task priorities can be changed at runtime

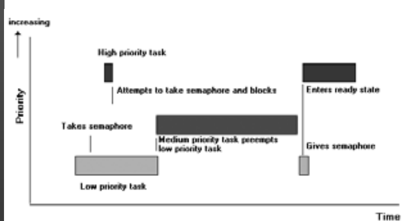
18

Priority

- The advantage to being able to change priority during runtime is that one can avoid occurrence of what is known as Priority Inversion
- *Priority inversion* occurs when the priority of a higher priority task is virtually reduced to the priority of a lower priority task
- This happens specially when the higher priority task is waiting for a resource which is in use by a lower priority task

19

Priority Inversion



20

Mutual Exclusion (mutex)

- The easiest way for tasks to communicate between each other is through shared data structures
- However, while exchanging data through shared resources makes communication easy, it poses a special challenge when more than one process may change a shared resource at the same time
- Therefore it is critical to ensure that a task has exclusive access to a shared resource before it changes any data
- **Mutex**

21

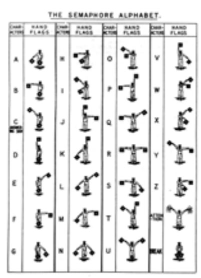
Mutex

- **The most important mechanism which must be provided at the CPU level in order to achieve mutual exclusivity is a *test-and-set-lock* (TSL) instruction**
- **Test-and-set-lock instructions come in various mnemonics and operations, but they all perform a single critical task: to allow an atomic operation which both tests (checks) a resource and sets (switches) its value in one operation**

22

Semaphores

- **There are other operations which can also be employed, such as disabling interrupts, disabling scheduling, and using semaphores**
- **Semaphores are of great use in real-time kernels. They allow operations on shared resources, and allow tasks to synchronize their operations**



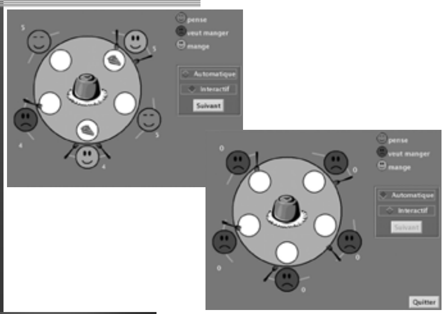
23

Semaphores

- **A semaphore is basically a flag which signals the right to use a shared resource**
- **A *binary semaphore* is either set or not set which then allows a task to use or have to wait to use a shared resource**
- **A *counting semaphore* on the other hand allows for more complex scenarios**

24

Dining Philosopher Problem



Scheduling

- A central piece of an operating system is the scheduler
- The scheduler maintains an overview of all tasks which are running or pending and decides which one to execute next
- There are many algorithms to determine which task to run next ranging from the simple *first-in-first-out* and *shortest-job-first*, to more complex priority-based algorithms found on real-time operating systems

26

Scheduling

- The oldest, simplest, fairest, and most widely used scheduling algorithm is round robin, where each process is assigned a time interval during which it is allowed to run
- Round robin scheduling makes the implicit assumption that all tasks are of the same priority.
- Priority based scheduling on the other hand takes the different priorities of tasks into account during scheduling, and changes them dynamically as may be necessary to increase system throughput

27

Deadlocks

- Since tasks share resources, a *deadlock* can occur if there is an interdependency between two tasks and their respectively locked resources
- Four conditions must be met for a deadlock to occur:
  - Mutual exclusion – a resource is already assigned
  - Hold and Wait – a process which has already been granted resources can seek new ones
  - No preemptive condition – resources previously granted cannot be forcibly taken away
  - Circular wait condition – there is a circular condition of at least two processes, each waiting for a resource held by the other one

28

Deadlock Recovery

- There are various means to recover from deadlocks
  - Recover by preemption: temporarily take away a resource from its current owner so that another process can continue executing. Not easy to do.
  - Recover through rollback: processes which hold the required resource are rolled back to a point in time before it requested the resource. All work done since the last *checkpoint* is lost.
  - Recover through killing the process: the simplest and crudest way of recovery

29

Interrupts

- Interrupts are primarily hardware mechanisms used to notify the processor that an asynchronous even has occurred
- When an interrupt occurs, the CPU saves the current context and jumps to the Interrupt Service Routine (ISR)
- Microprocessors can individually enable or disable interrupts, and assign different interrupt priorities to individual interrupt sources
- On a real-time system, interrupts should be disabled for as little as possible

30

Interrupt Latency

- By far one of the most important characteristics of a real-time kernel is the amount of time for which interrupts are disabled
- The longer interrupts are disabled, the longer the *interrupt latency* of the system
- Interrupt latency is the maximum time interrupts are disabled, plus the time it takes to start executing the first instruction of the ISR

31

Memory Management

- Parkinson's Law:

  Programs expand to fill the memory available to hold them!
- The part of the operating system which handles memory management is referred to as the *Memory Manager*
- The section of the processor which handles memory management is referred to as the *Memory Management Unit (MMU)*

32

Memory Management

- The memory management system is designed to make memory resources available to processes safely and efficiently
- The term *memory management* refers to the rules that govern mappings between the physical and virtual memory
- We are primarily concerned with two types of Memory Management:
  - Memory management over the *Logical* (virtual) address space
  - Memory management over the *Physical* address space (main memory)

33

Memory Management

- The memory management system is designed to make memory resources available safely and efficiently among threads and processes:
- It provides a complete address space for each process, protected from all other processes.
- It enables program size to be larger than physical memory.
- It allows efficient sharing of memory between processes.

34

Relocation

- Programs are *relocatable*, meaning that at run time the operating system will assign physical addresses to your program (relocate it) prior to loading it into the physical memory
- This also allows the operating system to swap the program out of memory and reload it at a different location at a later time
- Virtual Memory is essentially a technique which allows execution of a program which may not fit into the physical memory

35

Relocation

- Therefore the Operating System fakes a program into thinking that there is more memory space than is physically available to it, and Virtual Address is translated into Physical address

36