Need for OS

- OS in a computer or an embedded system provides a good solution to multitasking with zero fault tolerance to handle wide variety of inputs and output requirements.

- OS provides an efficient way to used memory and computational resources to reduce power

Need For OS

- OS system can know hardware well and can allocate, schedule and control resources in an optimal manner.
- OS can be made more powerful and robust to meet specific requirements of an embedded system or the application.
- Shall we say OS is the software component of the embedded system which is as important as the hardware?

Need for OS

•Embedded system without OS will be very difficult to program by a normal person.

• OS understands well all the resources available and can definitely optimize better to give a good performance.

• OS can be programmed to know the software and hardware capabilities as well as memory and I/O requirements.

• A computer can look at low speed (mouse) as well as high speed (image) inputs and at regular clock intervals and perform all expected operations as fast as possible. This would require a long program with all requirements at each such interval available!!

The Diversity of I/O devices

Device	Behaviour	Partner	Data Rate (KB/sec)
Keyboard	Input	Human	0.01
Mouse	Input	Human	0.02
PCI Bus	Input or Output	Machine	528000
Scanner	Input	Human	400.00
ISA Bus	Output or Input	Machine	16700
XGA Monitor	Input	Machine	60000
Graphics display	Output	human	60000
Laser printer	Output	Human	200.00
Modem	Input or Output	Machine	2.00-56.00
Network/ LAN	Input or Output	Machine	500 - 6000
Floppy disk	Storage	Machine	100.00
Optical disk	Storage	Machine	1000.00
Magnetic tape	Storage	Machine	2000.00
Magnetic disk	Storage	Machine	2000 - 100000

Four Main Functions of an OS

a) Process management

- process creation, process loading and execution control, the interaction of the process with signal events, process monitoring, CPU allocation and process termination

a) Inter process communication and synchronization

- Synchronization and coordination, deadlock and livelock detection and handling, process protection and data exchange mechanisms.

a) Memory management

- File creation, detection, reposition and protection

a) Input/Output management

- request and release subroutines for a variety of peripherals and read, write and reposition programs

OS in the Future

- In the future OS can reflect just like humans and adapt as needed.
- Can a software examine its own operation?

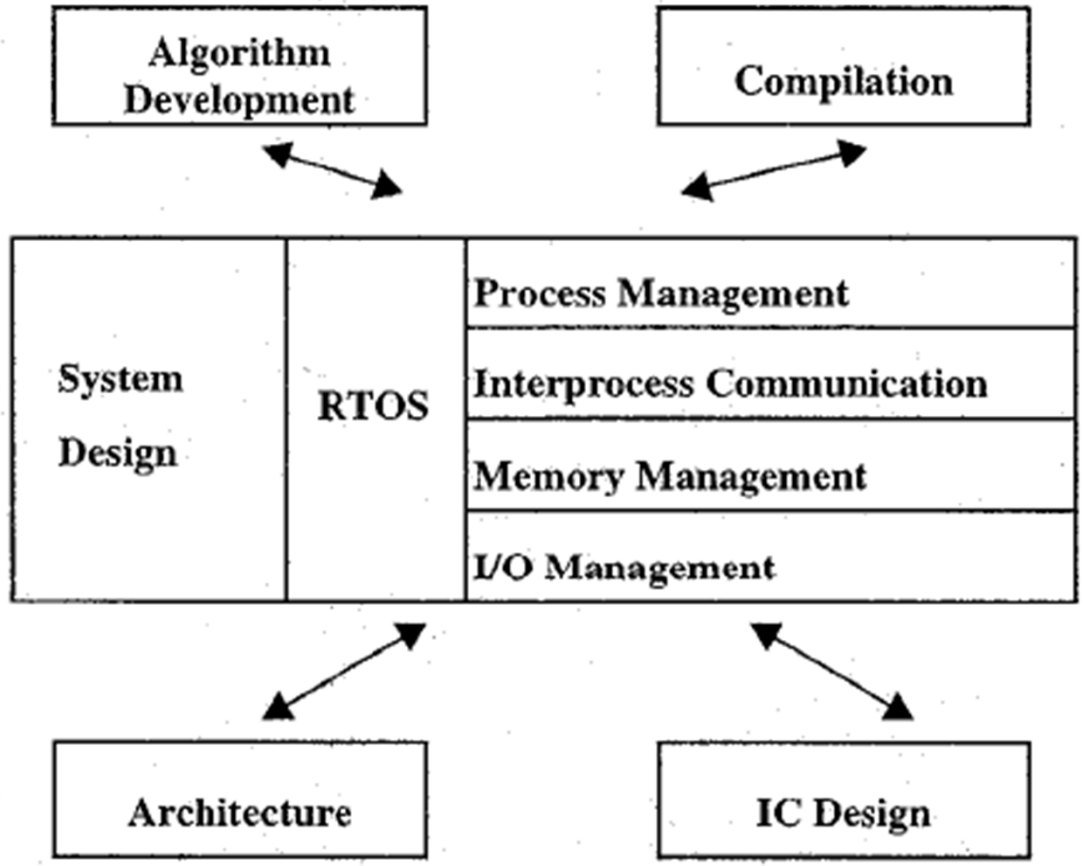
- Humans have an OS which we are yet to learn how it works!
- We don’t know much about the human brain and the way it functions. May be it does multitasking with varying degree of fault tolerance to extend or reduce brain life. So it may have an OS which we are yet to understand.

Real Time Operating Systems

The use of computers for control and monitoring of industrial processes has expanded greatly in recent years, and will expand even more dramatically in the near future with more and more smart and intelligent systems. Often, the computer or an embedded system used in such an application is shared between a certain number of time-critical control and monitor functions.

A Real Time Operating system (RTOS) is designed to handle input data within a guaranteed time. It helps in achieving correct timing behaviour of the system.

RTOS :The Backbone of New Design Process



RTOS Provides Interface and Isolation to Algorithm and Application Developers and compilation tools from one side to architecture and tools from other side.

3 Main States of an RTOS

The RTOS considers a process to be in one of three states: **waiting**, **ready**, and **executing**. In a uniprocessor system, only one process can be executing at any time. Some processes may be waiting for data or other events. Processes which are not blocked for external events but are not currently executing are considered ready. The transfer of execution from one process to another is called a **context switch**. To execute a context switch, the RTOS must save the state of the old process, determine what process will next obtain the CPU, and then set the CPU state to that process's state. Context switch overhead is non-trivial but often not a major factor in performance; scheduling policies, process partitioning, memory performance, and other factors are often more critical to obtaining good performance.

Real Time Operating Systems in the Market

- Integrated systems (PSOS)
- Wind River Systems (VxWorks)
- Microtec Research(VRTX)
- Microware Sysstems(OS-9)
- Hewlett-Packard (HP-RT)
- Lynx Real-Time Systems (LynxOS)
- Lucent(Inferno)
- Eonic Systems(Virtouso RTOS)
- Microsoft (Windows-CE)

Real Time Operating Systems in the Market

- Academic research dominated by scheduling techniques
- Industry focus on memory requirements, Context switch time and interrupt latency.
- RTOSes differ in terms

a) Scheduling priority levels
b) Pre-emption mechanisms
c) Scheduling algorithms (FIFO, round robin, rate monotonic, earliest deadline first)
d) File and I/O support
e) Sets of utility programs(linkers, compilers, library managers, debuggers and
f) more often interesting visual interfaces

Designing RTOS : Some Interesting Constraints

Suppose we are given a multiple facility system with two kinds of facilities--those which may be time shared and those which may be space shared. A set of programs to be executed so as to minimise the time for the whole set.

Each program i takes time ti to be executed when run alone and for this length of time uses fractions sik and rjj of space and time available at space facility k and time facility j respectively.

Let us further suppose that the adjacency constraints applies to every space facility; that is, for every program i and space facility k, it is necessary that the space fraction sik be allocated so that program i occupies a set of adjacent locations on facility k.

The Scheduling problem

A scheduling algorithm is a set of rules that determine the task to be executed at a particular moment.

The scheduling algorithms to be studied are pre-emptive and *priority driven* ones. This means that whenever there is a request for a task that is of higher priority than the one currently being executed, the running task is immediately interrupted and the newly requested task is started.

The specification of such algorithms amounts to the specification of the method of assigning priorities to tasks.

A Cooperative Scheduler

- A **cooperative** scheduler relies on the current process to give up the CPU before it can start the execution of another process. Cooperative multitasking is suitable for extremely simple systems, such as digital filters with purely periodic inputs, but is not used in sophisticated embedded systems.

A Static Priority Driven Scheduler

- A **static priority-driven** scheduler can **preempt** the current process to start a new process. The highest-priority ready process is always the currently executing process. Priorities are set before the system begins execution.

Dynamic Priority Driven Scheduler

- A **dynamic priority-driven** scheduler can redefine the process priorities at run time. The highest-priority ready process is still the currently-executing process, but since the RTOS can redefine priorities, the scheduling policy is embodied in the dynamic choice of priorities.

Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment

C. L. LIU *Project MAC, Massachusetts Institute of Technology*
AND

JAMES W. LAYLAND
Jet Propulsion Laboratory, California Institute of Technology

Journal of the Association for Computing Machinery, Vo|, 20, No. 1, January 1973, pp. 46-61.

ABSTR.CT. The problem of multi program scheduling on a single processor is studied from the viewpoint of the characteristics peculiar to the program functions that need guaranteed service. It is shown that an optimum fixed priority scheduler possesses an upper bound to processor utilization which may be as low as 70 percent for large task sets. It is also shown that full processor utilization can be achieved by dynamically assigning priorities on the basis of their current deadlines. A combination of these two scheduling techniques is also discussed.

The problem of multi program scheduling on a single processor : Assumptions

1. The requests for all tasks for which hard deadlines exist are periodic, with constant interval between requests.
2. Deadlines consist of run-ability constraints only--i.e, each task must be completed before the next request for it occurs.
3. The tasks are independent in that requests for a certain task do not depend on the initiation or the completion of requests for other tasks.
4. Run-time for each task is constant for that task and does not vary with time. Run-time here refers to the time which is taken by a processor to execute the task without interruption.

Rules of the Game of Scheduling

The *request rate* of a task is defined to be the reciprocal of its request period.

The *deadline* of a request for a task is defined to be the time of the next request for the same task.

Overflow: For a set of tasks scheduled according to some scheduling algorithm, we say that an *overflow* occurs at time t if t is the deadline of an unfulfilled request.

Feasible Scheduling: For a given set of tasks, a scheduling algorithm is *feasible* if the tasks are scheduled so that no overflow ever occurs.

The problem of multi program scheduling on a single processor

We will use J1, J2, • • • , Jm to denote m periodic tasks, with their request periods being T1, T2, ... , Tm and their run-times being C1, C2, ... , Cm, respectively.

Algorithms are priority driven and pre-emptive: meaning that the processing of any task is interrupted by a request for any higher priority task.

The specification of such algorithms amounts to the specification of the method of assigning priorities to tasks.

Some Assumptions

- All processes run periodically on a single CPU
- Context switching time is ignored
- There are no data dependencies between processes
- The execution time for a process is constant
- All deadlines are at the ends of their periods
- The highest priority ready process is always selected for execution.

Static Scheduling problem

A scheduling algorithm is said to be *static* if priorities are assigned to tasks once and for all. A static scheduling algorithm is also called a *fixed* priority scheduling algorithm.

Dynamic Scheduling problem

A scheduling algorithm is said to be *dynamic* if priorities of tasks might change from request to request.

Mixed Scheduling problem

A scheduling algorithm is said to be a *mixed scheduling algorithm* if the priorities of some of the tasks are fixed yet the priorities of the remaining tasks vary from request to request.

A Fixed Priority Scheduling Algorithm

Rate-Monotonic priority Assignment

Tasks with higher request rates will have higher priorities. Such an assignment of priorities is known as the *rate-monotonic priority assignment.*

It is a pre-emptive scheme. While executing a process if a higher priority process needs attention there will be context switching.

An Example for RMS

Consider a set of three tasks J1, J2 and J3 with T1 = 4, T2 =6 and T3= 12 with C1 = 1, C2 = 2 and C3= 3.

Give a RM Schedule.

J1 J2 J2 J3 J1 J3 J2 J2 J1 J3

0 1 2 3 4 5 6 7 8 9 10 11 12

An Example with context switching load

Consider a set of two tasks J1 and J2 with T1 = 5, T2 =10 and C1 = 3, C2 = 3.

Give a RM Schedule.

J1 J1 J1 J2 J2 J1 J1 J1 J2 J1 J1

0 1 2 3 4 5 6 7 8 9 10 11 12

Let the total time to initiate a process, including context switching and scheduling policy evaluation be 1 unit.

Now work out a schedule. Is there a feasible schedule?

J1 J1 J1 J2 J2 W J1 J1 J1 J2 J1 J1

0 1 2 3 4 5 6 7 8 9 10 11 12

Rate Monotonic Scheduling : Processor Utilization

Processor Utilization: $$U = \sum_{i=1}^{m} (C_i/T_i).$$

For a set of two tasks with fixed priority assignment, the least upper bound to the processor utilization factor is

$$U = m(2^{1/m} - 1).$$

For a set of m tasks with fixed priority order, the least upper bound to the processor utilization factor is

for large m, $$U \simeq \ln 2.$$

The Deadline Driven Scheduling Algorithm

• Priorities are assigned to tasks according to the deadlines of their current requests.

• A task will be assigned the highest priority if the deadline of its current request is the nearest, and will be assigned the lowest priority if the deadline of its current request is the furthest.

• Priorities will be recalculated at every completion of a process.

•At any instant, the task with the highest priority and yet unfulfilled request will be executed.

The Deadline Driven Scheduling Algorithm

For a given set of m tasks, the deadline driven scheduling algorithm is feasible if and only if

$$(C_1/T_1) + (C_2/T_2) + \cdots + (C_m/T_m) \leq 1.$$

The dynamic deadline driven scheduling algorithm is globally optimum and capable of achieving full processor utilization.

A Mixed Scheduling Algorithm

•A combination of the rate-monotonic scheduling algorithm and the deadline driven scheduling algorithm.

• Tasks 1, 2, • • • , k, the k tasks of shortest periods, be scheduled according to the fixed priority rate monotonic scheduling algorithm, and the remaining tasks, tasks k + 1,k + 2, • • • , m, be scheduled according to the deadline driven scheduling algorithm when the processor is not occupied by tasks 1, 2, • • • , k.

An Example

J1, J2 and J3 with execution times 1,1,2 and periods 3,4 and 5 respectively.

Fixed priority : C1 C2 C3 C1 C2 C3 C1 C3 C2 C1 C3 C3 C1 C2 C3 C1
Time slot : 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

Rate monotonic scheduling is not possible?

J1, J2 with execution times 2 and 1 with periods 4 and 12 respectively.

Fixed priority : C1 C1 C2 C1 C1 C1 C1 C1 C1 C2
Time slot : 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

Rate monotonic scheduling is possible

An Example: EDF

J1, J2 and J3 with execution times 1,1,2 and periods 3,4 and 5 respectively.

Fixed priority : C1 C2 C3 C1 C2 C3 C1 C3 C2 C1 C3 C3 C1 C2 C3 C1
Time slot : 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

Rate monotonic scheduling is not possible?

Deadline driven scheduling C2 C3
C1 C2 C3 C1 C2C1 C3 C1 C1
EDF : C1 C2 C3 C3 C1 C2 C1 C3 C3 C1 C2 C3 C3 C1 C2 C1
Time slot : 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

References

1. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment C. L. LIU *Project MAC, Massachusetts Institute of Technology* AND JAMES W. LAYLAND *Jet Propulsion Laboratory, California Institute of Technology*

3. Computers as Components, Principles of Embedded Computing System Design, Chapter 6, Wayne Wolf