Real-Time Operating Systems & Resource Management

Tei-Wei Kuo, Ph.D.
ktw@csie.ntu.edu.tw
Dept. of Computer Science & Information Engineering
National Taiwan University
Taipei, Taiwan, ROC

Remark: This set of slides comes from my class slides, my research work, slides generously provided by Dr. Jane W.S. Liu, and class slides contributed by authors of MicroC/OS-II (CMP Book).

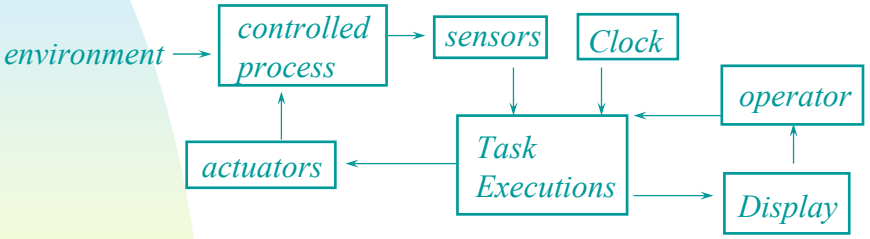
Contents

Overview

- The Purposes of Operating Systems
  - Convenience
  - Efficiency
- Characteristics of Many Real-Time/ Embedded Applications
  - More specific in their applications.
  - More drastic for their failures.

Overview

- A Typical Control System Example



- Rates - sensors & actuators, peripheral, control program
- Phases - takeoff, cruise, and landing, etc.

Overview

- Potential Timing Hazards:

```
Loop                                   ???Multiprogramming???
    ……
    Sensor();
    ……..
    computation……                                  Timer Granularity?
    ……..
    t = time();
    SleepTime := ReadyTime + PERIOD - t;
    ReadyTime = ReadyTime + PERIOD;
    Sleep(SleepTime);
EndLoop;                                            Real Sleep Time?
```

Loop Size?

Time Elapsed Here?

Overview

- General Concerns:
  - Could I verify the performance of my system?
  - How to avoid timing hazards?
  - Is there any good way in scheduling processes or allocating resources?

→ Understand your operating system and hardware, and use resources intelligently!

Contents

- Overview
- A General Architecture of Real-Time/ Embedded Operating Systems (RTOS's)
- Scheduling Strategies & System Analysis
- Process Synchronization over IPC
- Handling of Sporadic Events
- Summary

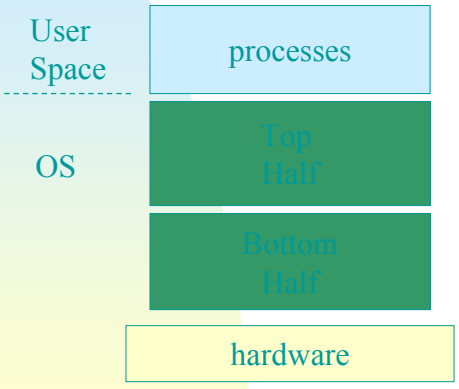
A General Architecture of RTOS's

- Various Requirements
  - Predictability – Verifiability
  - Reliability – Strictness of Deadline Violations
  - Reconfigurability – System Size and Functionality
  - Efficiency of System Components – Time Granularity, Threads, and Resource Management
  - Variable Models of Task Communication – Characteristics of Applications

A General Architecture of RTOS's

- Objectives in the Design of Many RTOS's
  - Efficient Scheduling Mechanisms
  - Good Resource Management Policies
  - Predictable Performance
- Common Functionality of Many RTOS's
  - Task Management
  - Memory Management
  - Resource Control, Including Devices
  - Process Synchronization

A General Architecture

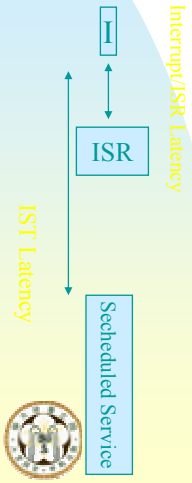


System calls such as I/O requests which may cause the releasing CPU of a process!

Timer expires to
- Expire the running process's time quota
- Keep the accounting info for each process

Interrupts for Services

A General Architecture



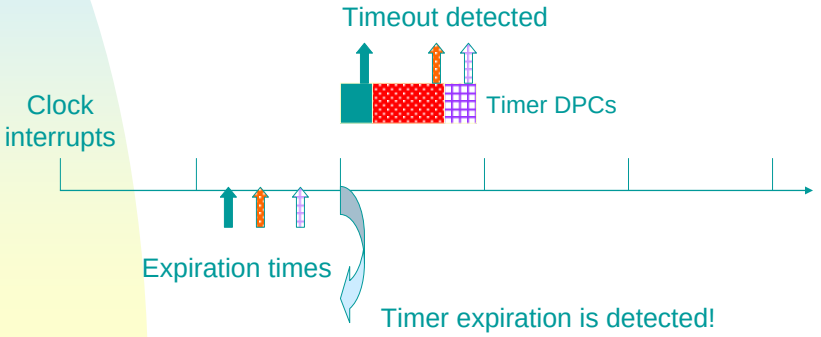
- 2-Step Interrupt Services
  - Immediate Interrupt Service
    - Interrupt priorities > process priorities
    - Time: Completion of higher priority ISR, context switch, disabling of certain interrupts, starting of the right ISR (urgent/low-level work, set events)
  - Scheduled Interrupt Service
    - Usually done by preemptable threads
  - Remark: Reducing of non-preemptable code, Priority Tracking/Inheritance (LynxOS), etc.

A General Architecture

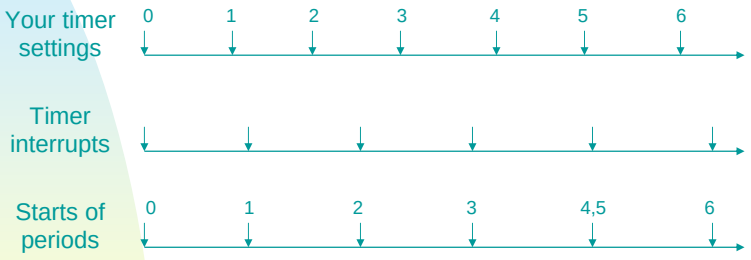
- Scheduler
  - A central part in the kernel
  - The scheduler is usually driven by a clock interrupt periodically, except when voluntary context switches occur – thread quantum?
- Timer Resolution
  - Tick size vs Interrupt Frequency
    - 10ms? 1ms? 1us? 1ns?
  - Fine-grained hardware clock

A General Architecture – Time Granularity

- For example, the timer granularity is equal to 1 ms or larger:



A General Architecture – For More Deterministic Periodicity



- Set the timer resolution to x.
- Round off all of timeout intervals to integer multiples of x!

A General Architecture – Timer

- Performance (time stamp) counter drifts.
- Counters on different processors are not synchronized.
- A thread reads the counter on the processor where it executes.

Consistent time stamps =

Readings of the same counter!

A General Architecture

- Memory Management
  - No protection for many embedded systems
  - Memory-locking to avoid paging
- Process Synchronization
  - Sources of Priority Inversion
    - Nonpreemptible code
      - Critical sections
    - A limited number of priority levels, etc.

Contents

- Overview
- A General Architecture of Real-Time/ Embedded Operating Systems (RTOS's)
- Scheduling Strategies & System Analysis
- Process Synchronization over IPC
- Handling of Sporadic Events
- Summary

Scheduling Strategies & System Analysis

- Why a process in my application does not meet its deadline?
- Factors:
  - Impacts from the executions of higher-priority processes – preemption cost
  - Lengthy execution time of the process
  - Blocking time from lower-priority processes – priority inversion

Scheduling Strategies & System Analysis

- Possible Questions:
  - How do I assign priorities to processes?
  - How are my processes scheduled by the OS?
  - How long is the blocking time/non-preemptable critical sections (from lower-priority processes or interrupts)?

→ Understand your schedulers

  - Fixed-Priorities or Dynamic Priorities
  - Preemptive or Non-Preemptive Scheduling

Scheduling Strategies & System Analysis Scheduling Strategy

- Major Components of a Scheduler
  - Priority Assignment Policy
    - The number of priority levels, e.g., 256?
    - Aging Effects?
  - Priority-Driven Scheduling Mechanism
    - Priority Queue
    - Thread Quantum?
    - Preemption Lock – Disabling of Preemption
    - etc.

Introduction to Real-Time Process Scheduling

- Q: Many theories and algorithms in real-time process scheduling seem to have simplified assumptions without direct solutions to engineers' problems. Why should we know them?
- A:
  - Provide insight in choosing a good system design and scheduling algorithm.
  - Avoid poor or erroneous choices.

Popular Process Model

- Periodic process
  - each periodic process arrives at a regular frequency - a special case of demand.
    - **r**: ready time, **d**: relative deadline, **p**: period, **c**: maximum computation time.
  - For example, maintaining a display
- Sporadic process
  - An aperiodic process with bounded inter-arrival time p.
  - For example, turning on a light
- Other requirements and issues:
  - process synchronization including precedence and critical sections, process value, etc.

Performance Metrics

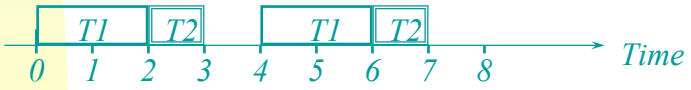
- Metrics for hard real-time processes:
  - Schedulability, etc.
- Metrics for soft real-time processes:
  - Miss ratio
  - Accumulated value
  - Response time, etc.
- Other metrics:
  - Optimality, overload handling, mode-change handling, stability, jitter, etc.
  - Combinations of metrics.

Basic Definitions

- Preemptive scheduling: allows process preemptions. (vs non-preemptive scheduling)
- Online scheduling: allocates resources for processes depending on the current workload. (vs offline scheduling)
- Static scheduling: operates on a fixed set of processes and produces a single schedule that is fixed at all time. (vs dynamic scheduling)
- Firm real-time process: will be killed after it misses its deadline. (vs hard and soft real-time)
- Fixed-priority scheduling: in which the priority of each process is fixed for any instantiation. (vs dynamic-priority scheduling)

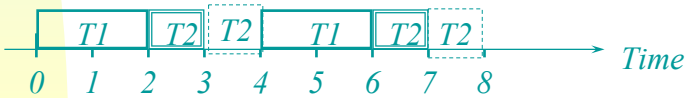
Rate Monotonic Scheduling Algorithm

- Assumptions:
  - all periodic fixed-priority processes
  - relative deadline = period
  - independent process - no non-preemptable resources
- Rate Monotonic (RM) Scheduling Algorithm
  - RM priority assignment: priority ~ 1/period.
  - preemptive priority-driven scheduling.
- Example: T1 (p1=4, c1=2) and T2 (p2=5, c1=1)



Rate Monotonic Scheduling Algorithm

- Critical Instant [1]
  - An instant at which a request of the process have the largest completion/response time.
  - An instance at which the process is requested simultaneously with requests of all higher priority processes
- Usages
  - Worst-case analysis
  - Fully utilization of the processor power
  - Example: T1 (p1=4, c1=2) and T2 (p2=5, c1=1)



[1] *Liu and Layland, "Scheduling Algorithms for multiprogramming in a hard real-time Environment," JACM, vol. 20, no. 1, January 1973, pp. 46-61.*

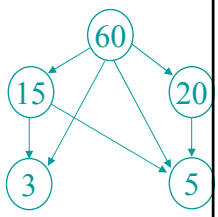
Rate Monotonic Scheduling Algorithm

- Schedulability Test:
  - A sufficient but not necessary condition
  - Achievable utilization factor α
    - of a scheduling policy P -> any process set with total utilization factor $\sum \frac{c_i}{p_i}$ no more than α is schedulable.
  - Given n processes, $\alpha = n\left(2^{1/n} - 1\right)$
- Stability:
  - Let processes be sorted in RM order. The ith process is schedulable if $\sum_{j=1}^{i} \frac{c_j}{p_j} \le i\left(2^{1/i} - 1\right)$
- An optimal fixed priority scheduling algorithm

Schedulability Tests – A Sufficient Condition

Theorem 0 [Liu&Layland 73]: A set of *n* periodic processes is schedulable if the total utilization factor of the process set is no larger than

$$n(2^{1/n} - 1)$$



Theorem 1 [Kuo, et al. 00]: Suppose that T_{i-1} is schedulable. Let *k* be the number of roots in T_i. If the total utilization factor of T_i is no larger than

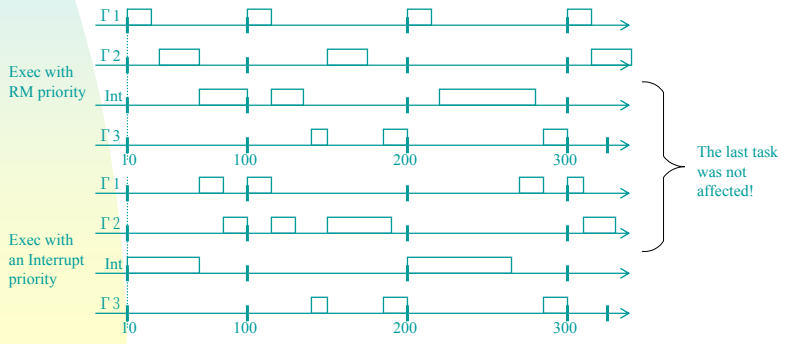
$$k(2^{1/k} - 1)$$

then T_i is schedulable.

τ_i	1	2	3	4	5
p_i	3	5	15	20	60
c_i	1	1	2	3	8
U_i	0.3333	0.5333	0.6666	0.8166	0.95

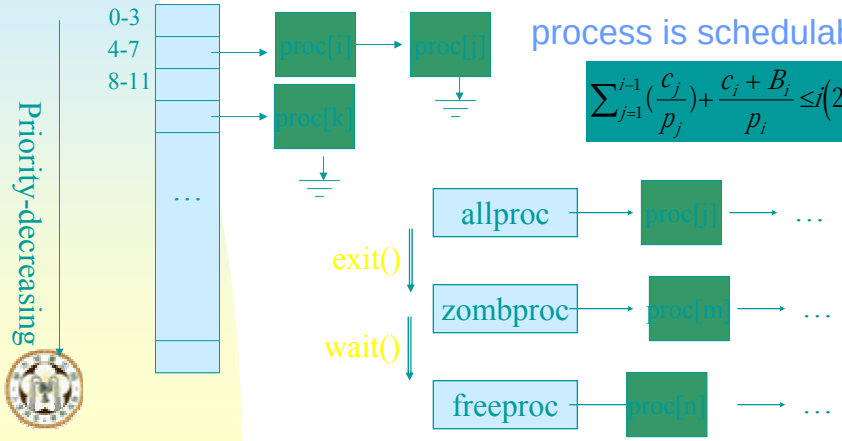
Schedulability Tests – Effects of Interrupts

Task Γ1 : C1 =20ms, P1 =100ms, U1=0.2
Task Γ2 : C2 =40ms, P2 =150ms, U2=0.267
Interrupt : Cint=60ms, Pint=200ms, Uint=0.3
Task Γ3 :C3 =20ms, P3 =350ms, U3=0.057



Schedulability Tests – Effects of Priority Mapping

- Ready Queue



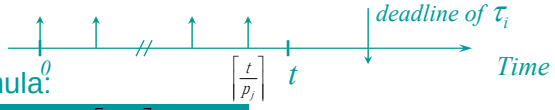
Let processes be sorted in RM order. The ith process is schedulable if

$$\sum_{j=1}^{i-1}\left(\frac{c_j}{p_j}\right)+\frac{c_i+B_i}{p_i}\le i\left(2^{1/i}-1\right)$$

Rate Monotonic Analysis – A Sufficient & Necessary Condition

- Rate Monotonic Analysis (RMA) [2]
  - Basic Idea:

    Before time t after the critical instance of process τ_i, a high priority process τ_j may request $c_j\left\lceil \frac{t}{p_j} \right\rceil$ amount of computation time.



  - Formula:

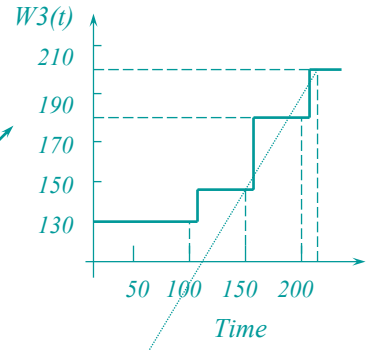
$$W_i(t) = \sum_{j=1}^{i} c_j \left\lceil \frac{t}{p_j} \right\rceil \le t \le d_i \quad \text{for some } t \text{ in } \{kp_j \mid j = 1,\ldots,i; k = 1,\ldots,\lceil p_i / p_j \rceil\}$$

  - A sufficient and necessary condition and many extensions...

[2] Sha, "An Intorduction to Rate Monotonic Analysis," tutorial notes, SEI, CMU, 1992

Rate Monotonic Analysis – A Sufficient & Necessary Condition

- A RMA Example:
  - T1(20,100), T2(30,150), T3(80, 210), T4(100,400)
  - T1
    - c1 <= 100
  - T2
    - c1 + c2 <= 100 or
    - 2c1 + c2 <= 150
  - T3
    - c1 + c2 + c3 <= 100 or
    - 2c1 + c2 + c3 <= 150 or
    - 2c1 + 2c2 + c3 <= 200 or
    - 3c1 + 2c2 + c3 <= 210
  - T4
    - c1 + c2 + c3 + c4 <= 100 or
    - 2c1 + c2 + c3 + c4 <= 150 or
    - ....

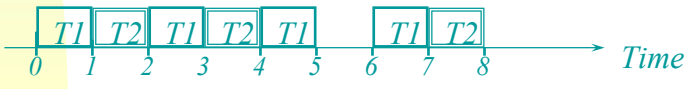


Rate Monotonic Scheduling Algorithm

- RM was chosen by
  - Space Station Freedom Project
  - FAA Advanced Automation System (AAS)
- RM influenced
  - the specs of IEEE Futurebus+
- RMA is widely used for off-line analysis of time-critical systems.

Earliest Deadline First Scheduling Algorithm

- Assumptions (similar to RM):
  - all periodic dynamic-priority processes
  - relative deadline = period
  - independent process - no non-preemptable resources
- Earliest Deadline First (EDF) Scheduling Algorithm:
  - EDF priority assignment: priority ~ absolute deadline. i.e., arrival time **t** + relative deadline **d**.
  - preemptive priority-driven scheduling
- Example: T1(c1=1, p1=2), T2(c2=2, p2=7)



Earliest Deadline First Scheduling Algorithm

- Schedulability Test:
  - A sufficient and necessary condition
  - Any process set is schedulable by EDF iff

$$\sum_{j=1}^{i} \frac{c_j}{p_j} \leq 1$$

- EDF is optimal for any independent process scheduling algorithms
- However, its implementation has considerable overheads on OS's with a fixed-priority scheduler and is bad for (transiently) overloaded systems.

Contents

Synchronization & IPC

- Why Inter-Process Communication (IPC)?
  - Exchanging of Data and Control Information!
- Why Process Synchronization?
  - Protect critical sections!
  - Ensure the order of executions!

Synchronization & IPC

- Critical Sections: A portion of code must be treated indivisibly.
  - To protect shared resources from corrupting due to race conditions.
  - Could be implemented by using interrupt enable/disable, semaphores, events, mailboxes, etc.

Synchronization & IPC

- Common Facility for IPC?
  - Shared Memory
  - Message Transmission
- Common Facility for Synchronization?
  - Semaphores
  - Signals

Synchronization & IPC

Example 1:

```
flag[i]=TRUE;
while flag[i]
    ;
…
flag[i]=FALSE;
```

Deadlock?

Example 2: (initially, lock=FALSE)

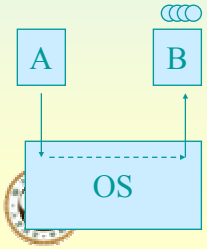
```
key=TRUE;
for(swap(lock,key); key==TRUE; )
  swap(lock,key) ;
…
lock=FALSE;
```

Livelocks?

- Shared Memory
  - Characteristics: High bandwidth and low latency, but very primitive!
  - Needs: Ways to synchronize its access to avoid racing conditions!
    - Potential deadlocks/ livelocks/blocking problems
    - e.g., semaphores, bakery algorithm, etc.

Synchronization & IPC

- Message Transmission
  - Characteristics: Simple & clean interface with various extensions: m:m communication, multi-machines
  - Needs: A priority-based message transmission/notification/processing mechanism
    - e.g., message priority, non-blocking library functions, notification of msg arrivals, servicing order of msgs (with respect to other threads in the system), etc.



Synchronization & IPC

- General Concerns:
  - How to enforce mutual exclusion?
    - We should not rely on OS scheduling to avoid race conditions!
  - How to process critical messages first?
    - There is a tight coupling between message processing and OS scheduling!
- After all, we want to manage the priority inversion problem!
  - How to let critical jobs be done with minimized interferences from less important jobs!

Synchronization & IPC

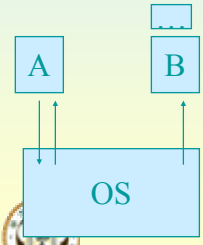
- Solutions??
  - Synchronization Methods
  - Priority-Driven Resource Scheduling Support!
- Popular Approaches
  - Semaphore-Based Synchronization?!
    - Signals are too slow.
  - Priority Inheritance?!
    - Ceiling?

Synchronization & IPC – Signals?

- The Design of Signal Mechanisms
  - Inform processes/threads of the occurrences of exceptions or events
- Posting of a signal to a process
  - An appropriate signal is added to the set of pending signals for the process.
- Delivering within the context of the receiver

```
if (sig = CURSIG(p))
  postsig(sig)
```

  - Signal handlers: user-mode routines



Synchronization & IPC – Signals

- Questions?
  - How fast can a signal be delivered?
    - Complicated actions done by OS
  - The length of a signal handler?
    - Signal handlers are executed prior to returning control to the user code.
- Real-Time Support
  - Application-defined signals
  - Queuing of signals while being blocked.
  - Signals are delivered in the priority order.
  - etc

Synchronization & IPC – Can we manage the priority inversion problem?

- What are the sources of priority inversion?
  - Synchronization and mutual exclusion
  - Nonpreemptable regions of code
  - FIFO of any other non-priority-based queues
  - Interrupts
  - A limited number of priorities

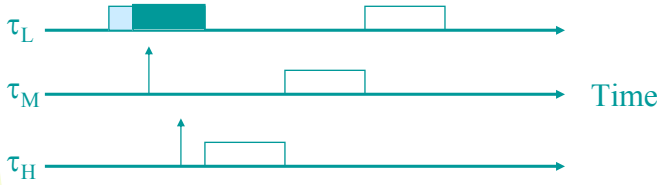
Synchronization & IPC – Can we manage the priority inversion problem?

- Popular Synchronization Methods
  - Nonpreemptable Critical Sections
  - Highest Locker's Priority
  - Priority Inheritance
  - Priority Ceiling

Synchronization Protocols

1. Nonpreemptible Critical Section

Critical sections are executed at an "infinitely" high priority!

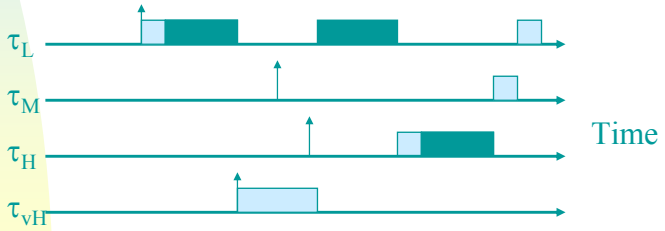


Note that τ_H & τ_M have no intention to enter a critical section!

Synchronization Protocols

2. Highest Locker's Priority Protocol

Execute critical section at the priority of the highest-priority task that may lock the semaphore(/resource); higher-priority tasks may preempt the critical section.

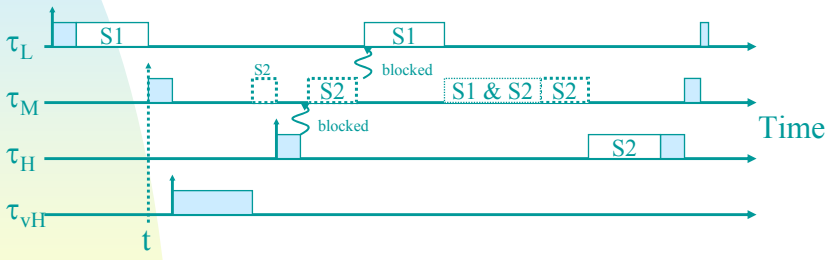


Note that τ_{vH} is no longer blocked by τ_L

Synchronization Protocols

3. Basic Inheritance Protocol (BIP) [Sha87]

Execute the critical section at the priority of the highest-priority task being blocked; higher-priority tasks may preempt the critical section.

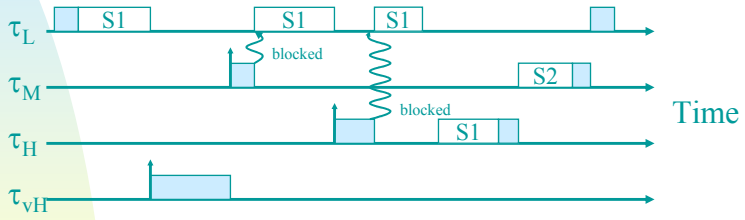


- Note that τ_M is no longer blocked until necessary.
- However, system may be deadlocked or have chained blocking!

Synchronization Protocols

4. Priority Ceiling Protocol [Sha87]

BIP + a "Priority Ceiling" rule about when to grant lock requests (see [Sha 87, 90])



- No deadlock & chained blocking at the cost of reducing the concurrency level of the system.

- Blocked-at-most-once.

Priority Ceiling Protocol

- Assumptions:
  - all periodic fixed-priority processes
  - relative deadline = period
  - Non-preemptable resources guarded by semaphores
- Basic Ideas and Mechanisms:
  - Bound the priority inversions by early blocking of processes that could cause them, and
  - Minimize a priority inversion's length by allowing a temporary rise in the blocking process's priority.
- Contribution of the Priority Ceiling Protocol
  - Efficiently find a suboptimal solution with a clever allocation policy, guaranteeing at the same time a minimum level of performance.

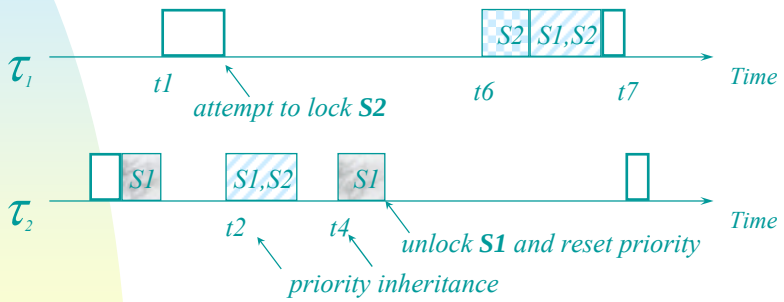
Priority Ceiling Protocol

- Pre-requirements: nested critical sections!
- Priority Ceiling Protocol (PCP):
  - Define a semaphore's priority ceiling as the priority of the highest priority process that may lock the semaphore.
  - Lock request for a semaphore is granted only if the requesting process's priority is higher than the ceiling of all semaphores concurrently locked by other processes.
  - In case of blocking, the task holding the lock inherits the requesting process's priority until it unlocks the corresponding semaphore. (Def: priority inheritance)

[1] *Sha, Rajkumar, and Lehoczky, "Priority Inheritance Protocols: an Approach to Real-Time Synchronization," IEEE Transactions on computers, Vol. 39, No. 9, Sept. 1990, pp. 1,175-1,185.*

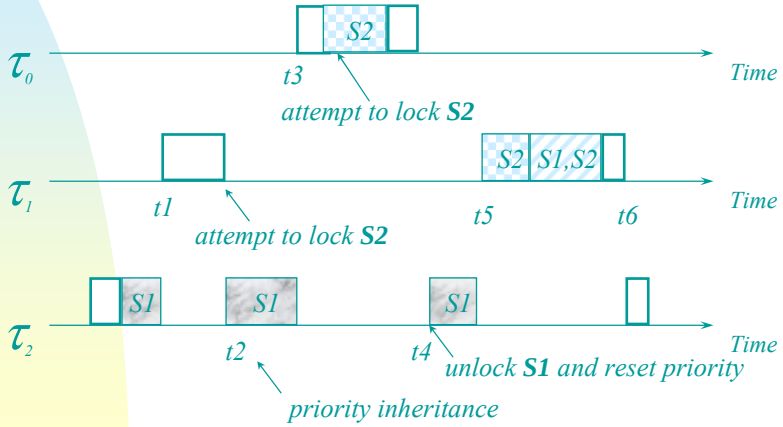
Priority Ceiling Protocol

- A PCP Example: deadlock avoidance



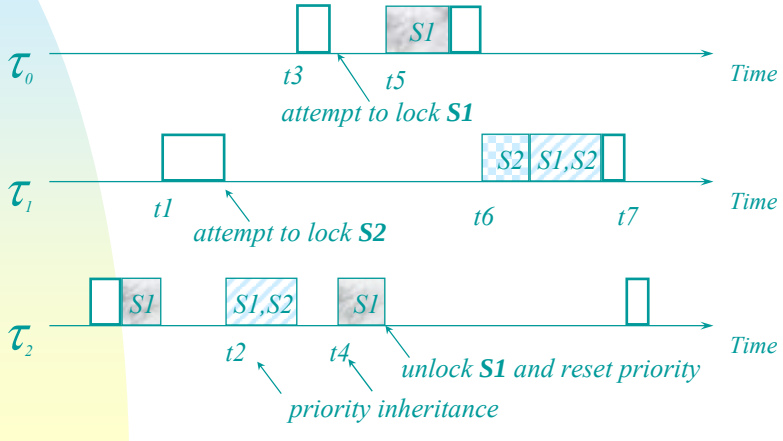
Priority Ceiling Protocol

- A PCP Example: avoid chain blocking



Priority Ceiling Protocol

- A PCP Example: one priority inversion



Priority Ceiling Protocol

- Important Properties:
  - A process is blocked at most once before it enters its critical section.
  - PCP prevents deadlocks.
- Schedulability Test of τ_i
  - worst case blocking time B_i - an approximation!

– $S_j = \{ S \mid semaphore\ S\ is\ accessed\ by\ \tau_j \}$
– $BS_i = \{ \tau_j \mid j > i\ \&\ Max_{(s\ in\ Sj)}\ (ceiling(s)) >= priority(\tau_i) \}$
– $B_i = Max_{(\tau j\ \ in\ BSi)}\ |critical\ section|$

  - Let processes be sorted in the RM priority order $\sum_{j=1}^{i-1}\left(\frac{C_j}{p_j}\right)+\frac{C_i+B_i}{p_i} \le I\left(2^{1/I}-1\right)$

Priority Ceiling Protocol

- Variations of PCP:
  - Stack Resource Policy - not permitted to start unless resources are all available.
    - multi-units per resource
    - dynamic and fixed priority assignments
  - Dynamic Priority Ceiling Protocol
    - extend PCP into an EDF scheduler.

3. Baker, "Stack-Based Scheduling of Real-Time Processes," J. Real-Time Systems, Vol. 3, No. 1, March 1991, pp. 67-99.
Chen and Lin, "Dynamic Priority Ceilings: A Concurrency Control Protocol for Real-time Systems," J. Real-Time Systems, Vol. 2, No. 4, Nov. 1990, pp. 325-340.

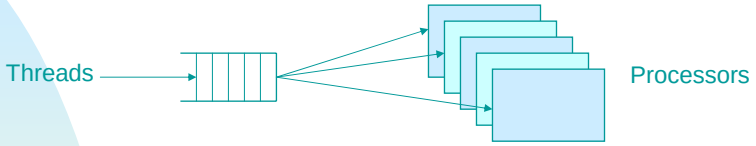
Summary of Synchronization Methods

	Blocked at Once	Deadlock Avoidance
Non-preemptible Critical Section	Yes	Yes
Highest Locker's Priority	Yes	Yes
Priority Inheritance	Bounded	No
Priority Ceiling	Yes	Yes

Remarks on Priority Inversion

- Symmetric Multi-Processor (SMP)
- Hyperthreading
- Deferred Procedure Call (DPC)

Priority Inversion in SMP



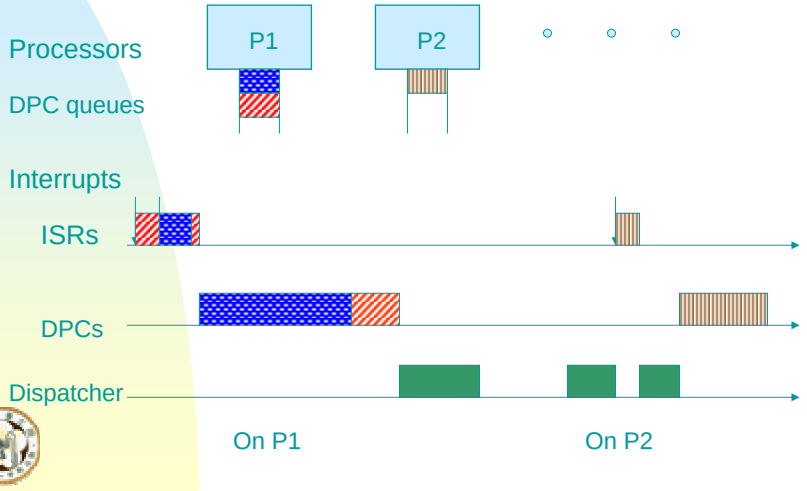
- Find-ready-thread function may not select the highest priority thread!
- Ready-thread function may not preempt the lowest priority thread!

Guarantee: The highest priority thread runs immediately after it becomes ready.

Process Inversion on Hyperthreading Processors

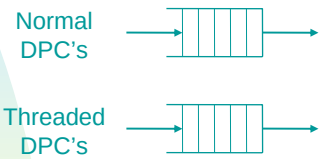
- Systems with hyperthreading is not equivalent to those with SMP!!
- Threads might share components!

DPC: Deferred Procedure Call (*Similar to tasklets in Linux*)



Threaded (Preemptive) DPC

- On each processor, there are:



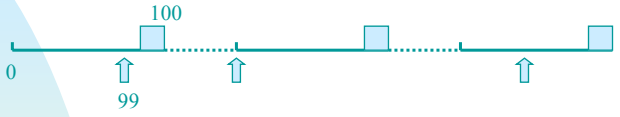
- Initialize each DPC either as a normal DPC or a threaded DPC.
- Use a new set of spinlock functions for threaded DPC's.

Contents

- Overview
- A General Architecture of Real-Time/ Embedded Operating Systems (RTOS's)
- Scheduling Strategies & System Analysis
- Process Synchronization over IPC
- Handling of Sporadic Events
- Summary

Scheduling Aperiodic Tasks

- Polling ~ Average Response Time = 50 units



- Interrupt Handler ~ Average Response Time = 1 unit

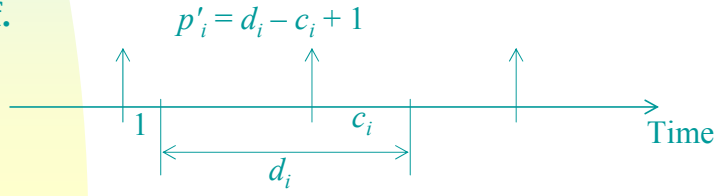
- Deferrable Server [Lehoczky87] ~ Average Response Time = 1 unit

  - An on-demand service type
  - When execution budget is used up, server execution drops to a lower (background) priority until the budgeted execution time is replenished.

Polling Services – Modeling of Sporadic Processes

Lemma 3[Mok, RTSS84]: Suppose we replace every sporadic process $\tau_i = (c_i, p_i, d_i)$ with a periodic process $\tau'_i = (c'_i, p'_i, d'_i)$ with $c'_i = c_i$, $p'_i = min(p_i, (d_i - c_i + 1))$, and $d'_i = c_i$. If the result set of all periodic processes can be successfully scheduled, then the original set of processes can be scheduled without *a priori* knowledge of the request times of the sporadic processes.

Proof.



Polling Services – Modeling of Sporadic Processes

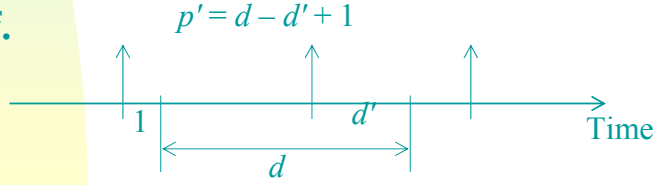
In general, we can replace each sporadic process $\tau_i = (c_i, p_i, d_i)$ with a periodic process $\tau'_i = (c'_i, p'_i, d'_i)$ if the following conditions are satisfied: [Mok, RTSS84]

(1) $d \geq d' \geq c$;

(2) $c' = c$;

(3) $p' \leq d - d' + 1$

Proof.



Deferrable Server (DS) [Lehoczky87]

- τ_1 as a DS server to preserve CPU bandwidth for a collection of aperiodic tasks.
- τ_1 has an entire period to use its C run-time and gets replenished at the beginning of each of its period.
- Theorem 3 [Lehoczky87]: For τ_1 as the highest priority DS server, and τ_2 … τ_n as periodic tasks, the achievable utilization factor is

$$U = U_1 + (n-1)[(\frac{U_1+2}{U_1+1})^{1/(n-1)} - 1] \qquad n \to \infty \quad U = U1 + \ln(\frac{U_1+2}{2U_1+1})$$

When U is minimized to 0.6518 when U_1 = 0.186.

J.P. Lehoczky, L. Sha, and J.K. Strosnider, "Enhancd Aperiodic Responsiveness in Hard Real-Time Environment," RTSS'87, pp261-270.

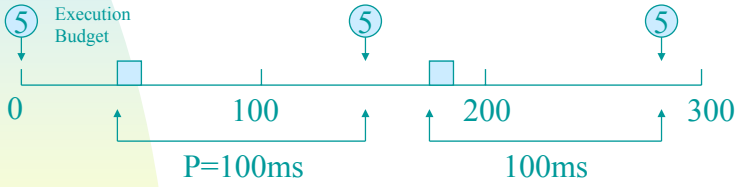
Remarks:

- Deferrable Servers:
  - –fixed execution budget
  - –replenishment interval
  - –priority adjusted to meet requirements.

    Note that the response time performance improves as the replenishment rate decreases because the execution budget increases and more services can be provided.
- Polling:
  - –From the scheduling point of view, polling converts the servicing of aperiodic events into an "equivalent" periodic tasks.
  - –Not an on-demand service type.
  - –As the rate of polling increases, the response time for polling approach improves.
- The rationale behind aperiodic servers:
  - –No "system" benefit to finish periodic work early!

Sporadic Server [Sprunt et. al. 90]

- Modeled as periodic tasks
  - fixed execution budget(c)
  - replenishment interval (p)



Replenishment occurs one “period” after the start of usage !

- Priority adjusted to meet requirements.

Sporadic Server [Sprunt et. al. 90]

- A sporadic server differs from a deferrable server in its replenishment policy:
  - A 100 msec deferrable server replenishes its execution budget every 100 msec , no matter when the execution budget is used.
- The affect of a sporadic server on lower priority tasks is no worse than a periodic task with the same period and execution time.

Sporadic Server [Sprunt et. al. 90]

- A sporadic server (SS) under the fixed-priority scheduling framework.
- Terms:
  - Ps: the priority at which the system is executing.
  - Pi: one priority level in the system.
  - Active: A priority level Pj is active is Pj <= Ps.
  - Idle: not active
  - RTi: replenishment time for SS executing at priority level Pi

Brinkley Sprunt, "Aperiodic Task Scheduling for Real-Time Systems," Ph.D. Thesis
Dept. of Electrical and Computer Engineering, Carnegie Mellon University, August 1990.

Sporadic Server [Sprunt et. al. 90]

- Rules:
  - Replenishment Time RTi
    - If SS has execution time available, and Pi becomes active at time t, then RTi = t + Pi.
    - If SS's execution time is exhausted, and SS's execution time becomes non-zero (is replenished) at time t and Pi is active, then RTi = t + Pi.
  - Replenishment Amount
    - Determined when Pi becomes idle or SS's execution time has been exhausted.
    - The amount is equal to the amount of server execution time consumed since the last time at which the status of Pi changes from idle to active.

Sporadic Server [Sprunt et. al. 90]

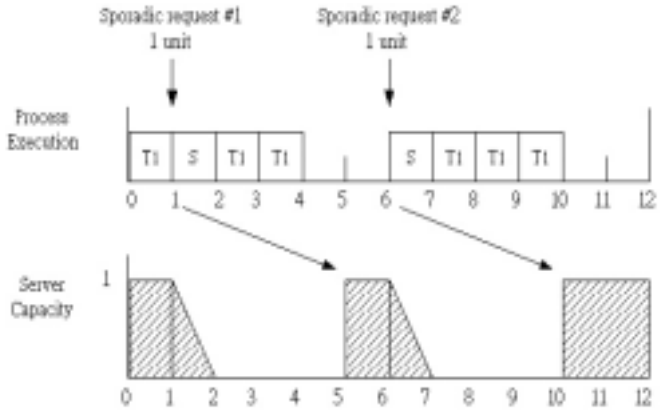
Important theorems:

- Theorem 1 [Sprunt90:p28]:Given a real-time system composed of soft-real-time aperiodic tasks and hard real-time periodic tasks, let the soft real-time aperiodic tasks be serviced by a polling server that starts at full capacity and executes at the priority level of the highest priority periodic task. If the polling server is replaced with a sporadic server having the same period, execution time, and priority, the sporadic server will provide high-priority aperiodic service at times earlier than or equal to the times the polling server would provide high-priority aperiodic service.

Sporadic Server [Sprunt et. al. 90]

- Theorem 5 [Sprunt90:p34]: A periodic task set that is schedulable with a periodic task Ti, is also schedulable if Ti is replaced by a sporadic server with the same period and execution time.
- Schedulability analysis of sporadic servers is equivalent to periodic tasks! -> overcome the penalty paid by the deferrable servers!
- Remark
  - In terms of server size, the sporadic server approach is better than the deferrable server approach.
  - Although the sporadic server approach claims low implementation overhead, it seems to be a little bit higher than the deferrable server approach.
  - If aperiodic services are requested very heavily , the differences between DS and SS will diminish.

A Sporadic Server Example



Contents

- Overview
- A General Architecture of Real-Time/ Embedded Operating Systems (RTOS's)
- Scheduling Strategies & System Analysis
- Process Synchronization over IPC
- Handling of Sporadic Events
- Summary

Some Ways to Better Predictability

- Eliminate major causes of unpredictability. (Poor hardware and drivers may block threads for a few hundred milliseconds.)
- Statically configure real-time components.
- Minimize the need for priority tracking across components and layers.
- Use synchronization protocols to control priority inversion.
- Verify drivers and applications.

Summary

- Understand your operating systems and hardware!
- There is no substitute for intelligent resource allocation methods!

Remarks on Multiprocessor Scheduling

郭大維 教授
ktw@csie.ntu.edu.tw
即時暨嵌入式系統實驗室
(Real-Time and Embedded Systems Laboratory)
國立臺灣大學資訊工程系

Preemptive Multiprocessor Scheduling

- Theorems of Mok in 1983
  - Goal:Understand the limitations of EDF.
  - Conditions:
    - different ready times.
  - Theorem 0: Earliest-deadline-first scheduling is not optimal in the multiprocessor case.
  - Example, T1(c=1,d=1), T2(c=1,d=2), T3(c=3,d=3.5), two processors.
  - Theorem 1: For two or more processors, no deadline scheduling algorithm can be optimal without complete a priori knowledge of deadlines, computation times, and process start times.

A.K. Mok, "Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment," Ph.D. Thesis, Dept. of Electrical Engineering and Computer science, MIT, May 1983.

Multiprocessor Anomalies

- Theorem of Graham in 1976.
  - Goal:Notice anomaly and provide better design.
  - Conditions;
    - A set of processes is optimally scheduled on a multiprocessor with some priority order, fixed execution times, precedence constraints, and a fixed number of processors.
  - Theorem 0: For the stated problem, changing the priority list, increasing the number of processors, reducing execution times, or weakening the precedence constraints can increase the schedule length.

R. Graham, "Bounds on the Performance of Scheduling Algorithms," Computer and Job Shop Scheduling theory, E.G. Coffman, ed., John Wiley and Sons, 1976, pp. 165-227.

Multiprocessor Anomalies

- An Example