

Analyzing Real-Time Systems

Reference: **Burns and Wellings, *Real-Time Systems and Programming Languages***

17-654/17-754: Analysis of Software Artifacts

Jonathan Aldrich



Real-Time Systems



- **Definition**
 - Any system in which the time at which output is produced is significant. This is usually because the input corresponds to some movement in the physical world, and the output has to relate to that same movement. The lag from input time to output time must be sufficiently small for acceptable timeliness – *Oxford Dictionary of Computing*
- **Examples make up 99% of microprocessors**
 - Industrial process control
 - Manufacturing
 - Communication – cell phones, etc.
 - Device controllers – thermostat, etc.

Hard and Soft Real Time



- **Hard**
 - Deadline must be met
- **Soft**
 - A task that may tolerate occasional missed deadlines
 - May be a limit on how often deadlines are missed, or how late a process may be
- **Firm**
 - May miss deadlines—but no benefit if result delivered after deadline
- **Real world**
 - Many systems have both hard and soft aspects
 - Many tasks are somewhere between hard and soft

10 April 2007

3

Analysis of Real-Time Systems



- **Many complex issues**
 - **Concurrency**
 - All the problems we studied before
 - **Priority inversion: can a low-priority task cause a high-priority one to miss its deadline?**
 - **Resource use**
 - **Especially memory resources**
 - **Safety criticality**
- **Most unique & critical: Timing**
 - Will a task be completed by its deadline?
 - **Two critical issues**
 - **How long does the task run in isolation?**
 - **When multiple tasks interact, will they each meet their deadline?**

10 April 2007

4

Timing and Memory Management



- Conventional malloc/free
 - Heap fragmentation
 - can take unbounded space (in theory)

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20

10 April 2007

5

Timing and Memory Management



- Conventional malloc/free
 - Heap fragmentation
 - can take unbounded space (in theory)

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20

10 April 2007

6

Timing and Memory Management



- Conventional malloc/free
 - Heap fragmentation
 - can take unbounded space (in theory)

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20

10 April 2007

7

Timing and Memory Management



- Conventional malloc/free
 - Heap fragmentation
 - can take unbounded space (in theory)

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20

10 April 2007

8

Timing and Memory Management



- Conventional malloc/free
 - Heap fragmentation
 - can take unbounded space (in theory)

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20

10 April 2007

9

Timing and Memory Management



- Conventional malloc/free
 - Heap fragmentation
 - can take unbounded space (in theory)

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20

10 April 2007

10

Timing and Memory Management



- Conventional malloc/free
 - Heap fragmentation
 - can take unbounded space (in theory)
 - example: 10x fragmentation overhead
 - malloc/free may not take bounded time
 - may need to search a free list for a block of the right size
- Use malloc/free designed for RT
 - e.g. TLSF (www.ocera.org)

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20

Timing and Memory Management



- Virtual memory
 - Allow graceful performance degradation
 - pages swapped to disk when application uses more than available memory
 - Page faults unpredictable
 - substantial delay
 - typically can't be used in RT systems
- Caches
 - Speed up average case memory access
 - Make worst-case access hard to predict
 - Hard RT systems: don't use cache
 - pay huge cost in performance
 - Typical RT systems: measure worst-case and add a safety factor

Timing and Memory Management



- Garbage collection (GC)
 - Automatically free unreachable data
 - Eliminates many leak problems
 - Can still have storage “leaks”
 - If you reference memory you aren’t going to use again
 - e.g. adding data objects to a global table
 - System pauses to collect garbage periodically
 - Typically unacceptable for real-time systems
- Real-time garbage collection
 - State of the art: IBM Metronome GC
 - Guarantees at least X% utilization of every time interval of length Y using space Z
 - Typical parameters
 - Y = 1 ms, X = 45%
 - Z = 2 * application space usage
- Conclusion: RT GC is feasible but must roughly double space, processor usage

10 April 2007

13

Memory Management Strategies



- Static allocation
 - pre-allocate all memory in static data blocks
 - pro: predictable memory usage
 - con: wastes a lot of space
- Stack allocation
 - allocate memory on stack as functions are called
 - pro: predictable memory usage
 - must analyze all call chains
 - must have no recursive functions (generally true of RT systems)
 - con: still wastes space when memory usage doesn’t fit stack discipline
 - con: dangling pointers are possible
 - but static analysis can find these
- Region-based allocation – used in Real Time Java
 - allocate memory in regions
 - regions are freed when no longer needed
 - pro: efficient in time and memory usage
 - when memory is used in predictable chunks with a predictable lifetime
 - con: dangling pointers possible
 - but static analysis can find these
- Real-Time GC
 - pro: nicest programming model
 - con: constant % of wasted space and time (but in some cases the space constant may be smaller than other techniques)

10 April 2007

14

Real-Time Process Model



- Fixed set of processes ($a-z$)
- Processes are periodic with period T
- Processes have deadlines D equal to period T
- Processes have fixed worst-case execution time C
- Processes are independent
- System/context switch overhead is ignored (assumed to be zero)
- Some of these assumptions will be relaxed later

Scheduling



- How to schedule the execution of processes so that each meets its deadline
- Multiple approaches
 - Cyclic Executive
 - Run processes in a fixed order
 - Fixed-Priority Scheduling
 - Order processes by priority
 - Always run highest priority process
 - Earliest-Deadline First Scheduling
 - Dynamically pick process closest to its deadline
- Complicating factors
 - Computing Worst-Case Execution Time
 - Sporadic processes
 - Interaction between processes and blocking

Computing Worst Case Execution Time



- Compile to machine code
- Divide code into basic blocks
 - straightline code with no jumps into or out of the middle
- Compute worst case time for each block
 - analytic: requires detailed model of processor and memory hierarchy
 - cache, pipelines, memory wait states, etc.
 - measurement: may not be worst case
 - may add engineering safety factor
- Collapse control flow graph
 - choice between blocks → choose max time
 - loops → use bound on maximum loop iterations
 - specialized knowledge may tighten bounds
 - taking if branch precludes taking a later else branch
 - branch may be taken only a limited number of times in loop
- Safer to measure basic blocks & combine with analysis than to measure worst case of entire program

10 April 2007

17

Worst Case Execution Time Example



```
for i = 1 to 100 do
  if (i % 2)
    then A (cost 10)
    else B (cost 30)
  if (i % 10)
    then C (cost 100)
    else D (cost 10)
end
```

cost of test = 1

10 April 2007

18

Cyclic Executive

[example from Burns & Wellings]



- Assume all deadlines are multiples of the *minor cycle time*

- 25 ms in the example

Idea

- If a task must be completed within n cycles, do $1/n$ of the work each cycle
 - benefit: if schedule is possible this one will work
 - cost: have to break up processes
 - expensive to add context-switches
 - bad for software engineering

Process	T	C
a	25	10
b	25	8
c	50	5
d	50	4
e	100	2

```

loop
  wait()
  a()          10.0
  b()          8.0
  half_of_c() 2.5
  half_of_d() 2.0
  quarter_of_e() 0.5
end loop      // total 23 ms
    
```

10 April 2007

19

Cyclic Executive, Revised

[example from Burns & Wellings]



- Improved idea
 - If several tasks must be split over several minor cycles, divide them among the minor cycles
 - Repeat this *major cycle*
- This is the bin-packing problem
 - NP complete, therefore solved heuristically
- If it works, use it
 - Simple, effective
- But sometimes inapplicable
 - Deadlines are not a multiple of any reasonable minor cycle time
 - Very long processes
 - have to be broken up

Process	T	C
a	25	10
b	25	8
c	50	5
d	50	4
e	100	2


```

loop
  wait()
  a() 10
  b() 8
  c() 5
  wait()
  a() 10
  b() 8
  d() 4
  e() 2
  wait()
  a() 10
  b() 8
  c() 5
  wait()
  a() 10
  b() 8
  d() 4
    
```

10 April 2007

end

20

Fixed Priority Scheduling (FPS)



- Assign each process a fixed priority
 - Based on time requirements, not importance
- Always run the highest priority active process
 - Processes released on schedule, according to period T
 - Run highest priority process first
 - When it completes run next priority process
 - New high-priority processes may preempt running processes of lower priority
 - non-preemptive schemes are possible but are more complicated to analyze

10 April 2007

21

Deadline Monotonic Priorities

[example from Burns & Wellings]



- Shortest deadline first
 - If $D_i < D_j$ then $P_i > P_j$
- | Process | T/D | P |
|----------------|------------|----------|
| a | 25 | 5 |
| b | 60 | 3 |
| c | 42 | 4 |
| d | 105 | 1 |
| e | 75 | 2 |
- Optimality theorem: *if any process set can be scheduled using preemptive priority-based scheduling, deadline monotonic priority ordering will schedule it. (Leung and Whitehead, 1982)*

10 April 2007

22

Timeline Schedulability Test



- How do you know if a set of processes is schedulable given a set of priorities?
 - i.e. will all processes meet their deadlines?
 - Theorem about deadline monotonic priorities only says that approach will find a schedule *if one exists*...but it may not exist!
- Timeline test (for deadline $D = \text{period } T$)
 - Assume all processes are released together
 - This is the maximum possible load on the processor
 - Known as the critical instant
 - Track process execution until all processes have passed their deadline
 - Theorem: *If a set of processes are tracked from the critical instant until the process with the longest period is complete, and all processes meet their deadlines, then all deadlines will be met thereafter (Liu and Layland, 1973)*
 - assumes deadline $D = T$

10 April 2007

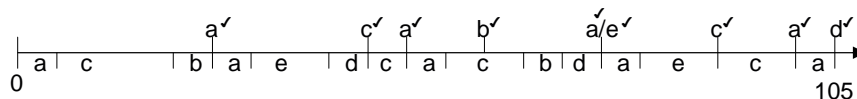
23

Timeline Schedulability Example



Process	T/D	C	P
a	25	5	5
b	60	5	3
c	45	15	4
d	105	10	1
e	75	10	2

Process Deadlines



Process Execution

All deadlines are met

10 April 2007

24

Earliest Deadline First (EDF) Scheduling



- Always run the process with the earliest deadline
- Theorem: *As long as the sum of the utilizations of the processes is less than or equal to 100%, EDF can schedule a process set. (Liu and Layland, 1973)*
 - Utilization of process $i = C_i/T_i$

10 April 2007

25

Example: EDF vs. FPS

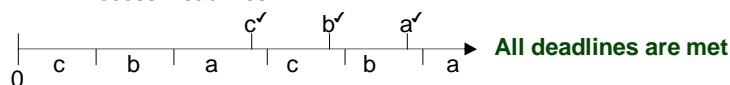
[example from Burns & Wellings]



Process	T/D	C	P	Util
a	50	12	1	24%
b	40	10	2	25%
c	30	10	3	33%

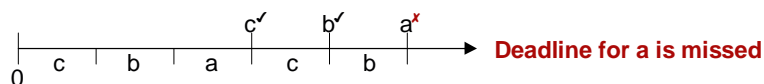
Total utilization 82%

EDF Process Deadlines



EDF Process Execution

FPS Process Deadlines



FPS Process Execution

10 April 2007

26

Discussion: EDF vs. FPS [from Burns & Wellings]



- EDF can schedule process sets which FPS cannot
 - fixed priority is not flexible enough
 - EDF makes a dynamic choice based on deadlines
- Why not always use EDF?
 - FPS is easier to implement and more efficient
 - Schedule based on purely static priorities
 - Dynamic choices require complex run-time system and higher overhead
 - Incorporating processes without deadlines is easier in FPS
 - Just give them a priority; no need to set artificial deadline
 - Can use other factors (e.g. importance) for priority in FPS
 - EDF forces all scheduling decisions to be based on deadlines
 - FPS is more predictable in overload situations
 - Lower priority processes will miss deadlines first
 - In EDF cascade of processes may miss deadlines – domino effect

10 April 2007

27

Sporadic Processes



- What about processes that do not arrive according to a fixed periodic schedule?
- Fixed priority scheduling
 - Set T to minimum inter-arrival interval
 - Sufficient to guarantee responsiveness
 - $D=T$ may be unrealistic
 - Allow $D < T$
 - Set priorities based on D

10 April 2007

28

Soft Processes



- Sporadic processes
 - Worst case may be much worse than average-case
 - May be unrealistic (and unnecessary) to make all deadlines
- Distinguish hard and soft processes
- Scheduling rules [Burns & Wellings]
 - All processes should be schedulable using average execution times and average arrival rates
 - All hard real-time processes should be schedulable using worst-case execution times and worst-case arrival rates of all processes (including soft)

10 April 2007

29

Servers



- Typically sporadic soft real time processes have low priority
- Means they may often miss their deadlines
 - Even when resources are available to meet their deadlines
- Deferrable Server (Lehoczky et al., 1987)
 - Schedule all hard processes
 - Add a server process
 - Highest priority
 - Some period T_s
 - Maximum capacity C_s such that system is still schedulable
 - When an sporadic process arrives
 - Run as part of server process until complete or server has used C_s capacity
 - After time T_s server can continue executing the sporadic process

10 April 2007

30

Blocking and Priority Inversion



- Unrealistic assumption: processes are independent
- Typical case: processes access shared resources
- Resources protected by locks
- Priority inversion (Lauer and Satterwaite, 1979)
 - Low-priority process acquires a lock
 - High-priority process has to wait for low priority process to release the lock
 - Could wait a long time, especially if medium-priority processes preempt the low-priority process

10 April 2007

31

Priority Inversion Case: Mars PathFinder



[Paulo Marques]



The Mars PathFinder would run ok for a while and, after some time, it would stop and reset.

The watch-dog timer was resetting the system because of some unknown reason...

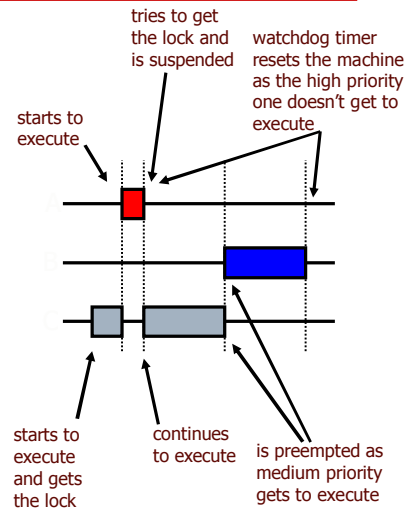
10 April 2007

32

The Mars PathFinder Problem [Paulo Marques]



- **Priority Inversion Problem**
 - Low priority thread locks a semaphore.
 - High priority thread starts to execute and tries to lock the same semaphore. It's suspended since it cannot violate the other thread's lock.
 - Medium priority threads comes to execute and preempts the low priority thread. Since it doesn't need the semaphore, it continues to execute.
 - Meanwhile the high priority thread is starving. After a while, the watchdog timer detects that the high priority thread is not executing and resets the machine.



10 April 2007

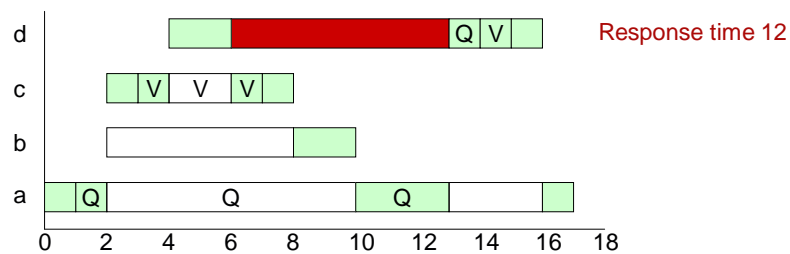
33

Priority Inversion Example

[example from Burns & Wellings]



Process	Priority	Execution	Release Time
a	1	EQQQE	0
b	2	EE	2
c	3	EVVE	2
d	4	EEQVE	4



10 April 2007

34

Priority Inheritance



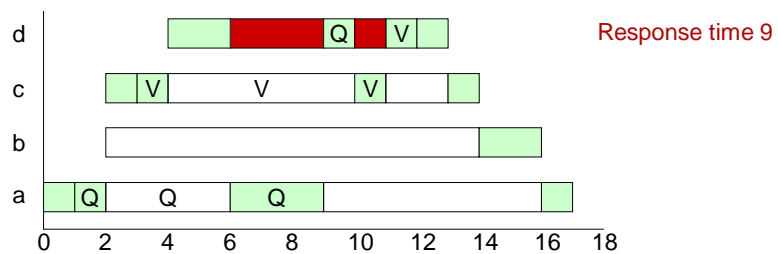
- High priority process P is blocked on resource R
- R is currently held by low-priority process Q
- Priority Inheritance (Cornhill et al., 1987)
 - Boost priority of Q to that of P
 - Ensures that Q will finish quickly so that P can proceed

Priority Inheritance Example

[example from Burns & Wellings]



Process	Priority	Execution	Release Time
a	1	EQQQE	0
b	2	EE	2
c	3	EVVE	2
d	4	EEQVE	4



Priority Ceilings



- Can we do better than Priority Inheritance?
 - In the example, can we ensure process d is not blocked twice?
- Priority ceiling of resource R
 - The maximum priority of the processes that use R
- Dynamic priority of process P
 - The maximum of P's static priority and the ceiling values of any resources it has locked
 - Intuition: if I've locked a resource R that some higher priority process Q might need, this rule ensures that no process with lower-priority than Q can possibly run and acquire some other resource R', so that Q would have to wait for two processes to acquire R and R'

10 April 2007

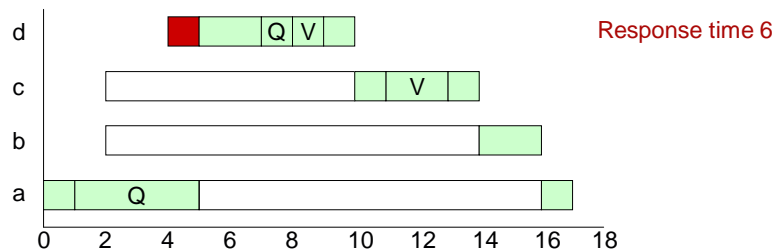
37

Priority Ceiling Example

[example from Burns & Wellings]



Process	Priority	Execution	Release Time
a	1	EQQQE	0
b	2	EE	2
c	3	EVVE	2
d	4	EEQVE	4



10 April 2007

38

Benefits of Priority Ceilings

[Burns & Wellings]



- A high priority process can be blocked at most once by lower-priority processes
- Deadlocks are prevented
 - on single-processor systems
- Transitive blocking is prevented
- Mutually exclusive access to resources is ensured

10 April 2007

39

Summary: Analysis of Real-Time Systems



- Resource use is challenging
 - Memory typically allocated statically or on stack
 - RT Java: regions
 - Future: real-time garbage collection
- Scheduling
 - Simple case: cyclic executive
 - General case: fixed priority scheduling
 - Assign priorities according to deadlines
 - Timelines used to check feasibility
 - Use priority inheritance to avoid priority inversion problems

10 April 2007

40