جامعة دمشق
كلية الهندسة المعلوماتية
قسم النظم والشبكات الحاسوبية
السنة الخامسة

نظم الزمن الحقيقي
Real time systems

د. مياد جابر

مقدمة

نظام الزمن الحقيقي؟

- تعريف١: هو نظام يستجيب للقيود الصريحة والمحددة المتعلقة بزمن الاستجابة وإلا فإن النظام قد يتعرض لعواقب وخيمة عند الفشل
- تعريف٢: هو نظام يعمل بشكل صحيح منطقياً بالاعتماد على صحة مخرجاته و احترامها للقيود الزمنية
- تعريف٣: هو نظام يستجيب لقيم إدخال خارجية ضمن فترة زمنية محددة ومعرفة بشكل مسبق

خصائص نظم الزمن الحقيقي

- الاستجابة ضمن الوقت المطلوب للأحداث الطارئة
- درجة عالية من الجدولة الزمنية: يجب تحقيق المتطلبات الزمنية للنظام في حالات الطلب العالي على الموارد
- استقرار النظام في حالات التحميل الزائد العابرة: في هذه الحالة وعندما يصبح من غير الممكن تنفيذ كل الأحداث ضمن الحدود الزمنية المفترضة، يجب ضمان إمكانية تنفيذ بعض المهام المحددة بصفتها مهام حرجة ضمن حدودها الزمنية deadlines

مقارنة

	نظام غير متعلق بالزمن الحقيقي	نظام زمن حقيقي
القدرة	إنتاجية عالية	الجدولة الزمنية: قابلية النظام لاحترام الحدود الزمنية النهائية
الاستجابة	معدل سريع للاستجابة	ضمان الحد الأقصى للتأخير
الحمل الزائد	العدالة في التوزيع	احترام الحدود الزمنية الهامة

المجالات التطبيقية لنظم الزمن الحقيقي

- نظم الاتصالات
- نظم التحكم بالمركبات
- مخدمات الوسائط المتعددة
- نظم معالجة الاشارة
- نظم الرادار
- نظم التصنيع المؤتمتة
- النظم الدفاعية
- الطائرات
- التحكم بحركة الملاحة الجوية
- نظم الملاحة الآلية
- الأقمار الصناعية
- نظم التحكم بالمفاعلات النووية

نماذج نظم الزمن الحقيقي

- نظم التنفيذ الحلقي: Cyclic executives وتسمى أيضاً نظم الخطوط الزمنية timelines أو النظم المبنية على الإطارات frame-based systems
- النظم المقادة بالأحداث: Event-driven systems بنوعيها الدورية periodic وغير الدورية aperiodic
- النظم الأنبوبية: Pipelined
- نظم مخدم/عميل: Client-server
- نظم آلة الحالة: State machine

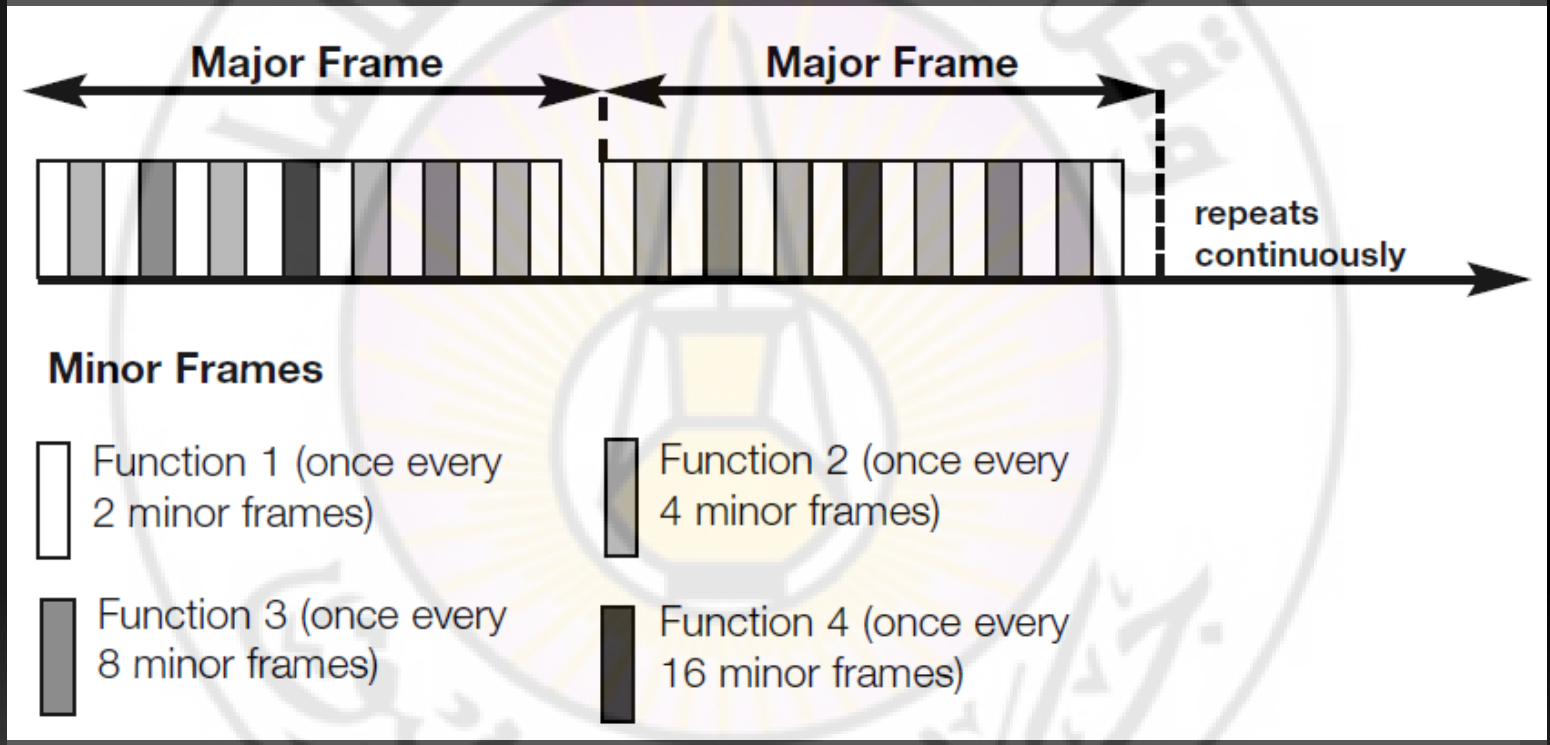
نظم التنفيذ الحلقي١
Cyclic executives

- تتكون الحلقة التنفيذية من سلسلة من المهام المكررة بشكل مستمر وتعرف بـ الإطارات الرئيسة
- يتكون كل إطار رئيسي من عدد من الفواصل الزمنية الأصغر وتعرف بـ الإطارات الثانوية minor frames
- يتم جدولة المهام ضمن الإطارات الثانوية
  - يقوم الخط الزمني باستخدام مؤقت من أجل إطلاق trigger مهمة في كل إطار ثانوي

نظم التنفيذ الحلقي٢
Cyclic executives

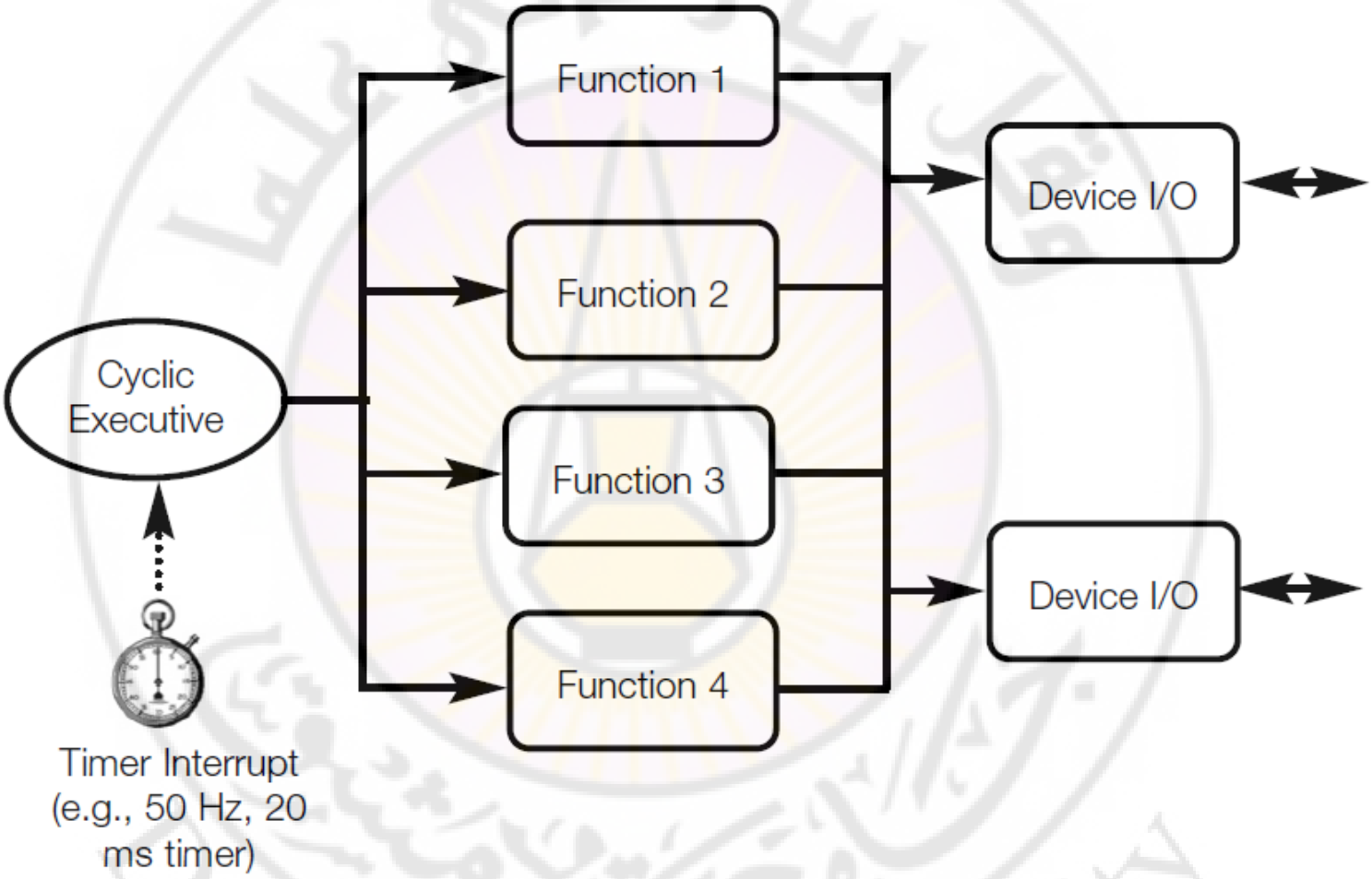
- مجموعة الإطارات الثانوية غير المكررة تشكل الإطار الرئيس
- يتم تنفيذ العمليات على شكل إجرائيات procedures حيث يتم وضعها ضمن قائمة معرفة مسبقاً وتغطي جميع الإطارات الثانوية
- عندما يبدأ تنفيذ حلقة ثانوية تقوم المهمة الخاصة بالمؤقت باستدعاء كل إجرائية من القائمة
- التنافس غير وارد في هذه الحالة: يتم تقسيم العمليات الكبيرة لتناسب الإطارات الثانوية

CYCLIC EXECUTIVES

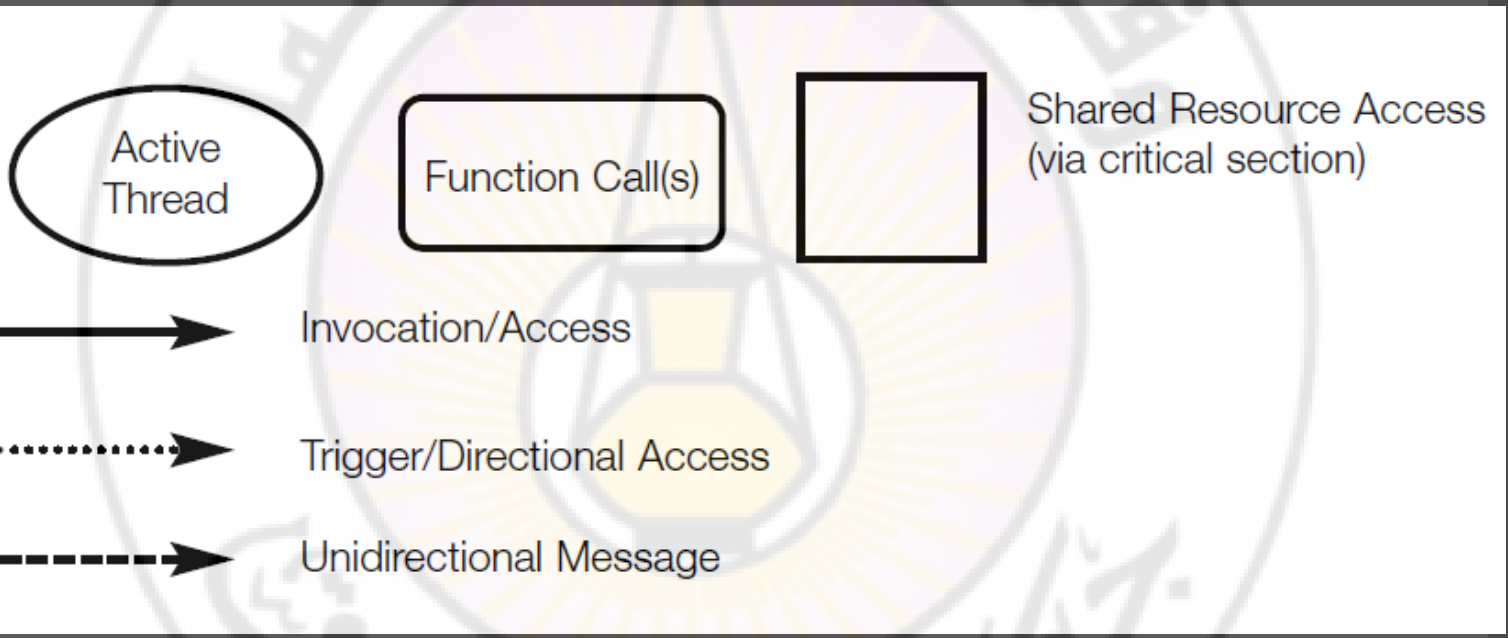


A sample cyclic executive نموذج تنفيذ حلقي

CYCLIC EXECUTIVES



A sample cyclic executive نموذج تنفيذ حلقي


Active Thread
Function Call(s)
Shared Resource Access (via critical section)
Invocation/Access
Trigger/Directional Access
Unidirectional Message

KEY

key

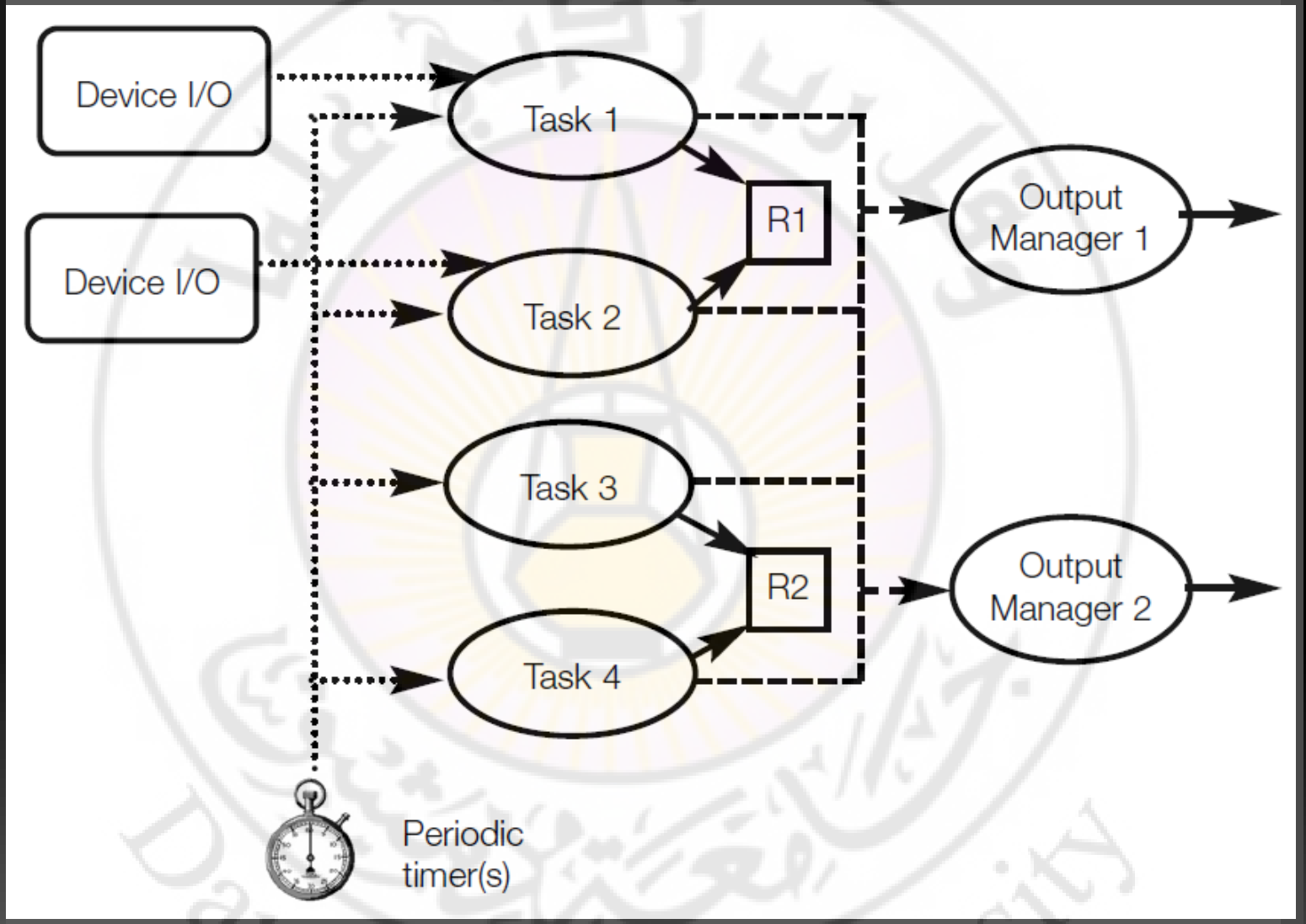
النظم المقادة بالأحداث١
Event-driven systems

- تستخدم هذه النظم أجهزة إدخال/إخراج زمن حقيقي أو أحداث مؤقت من أجل إطلاق trigger المهام المجدولة.
- يمكن إدارة الأولويات للمهام كما يلي:
  - وفق القيود الزمنية: المعدل rate أو الحد الزمني النهائي deadline
  - وفق الأهمية الدلالية semantic importance

النظم المقادة بالأحداث٢
Event-driven systems

- يتطلب التنافس الناتج عن هذه النظم ضمان عملية التزامن synchronization
  - في حالة الاستجابة الممكن التنبؤ بها يجب على آلية التزامن تفادي تغيرات الأولوية غير المحدودة
  - من أجل المحافظة على استجابة ممكن التنبؤ بها يجب على الأحداث غير الدورية الالتزام باستخدام الحدود

EVENT-DRIVEN SYSTEMS



نظام مقاد بالأحداث Event-driven system

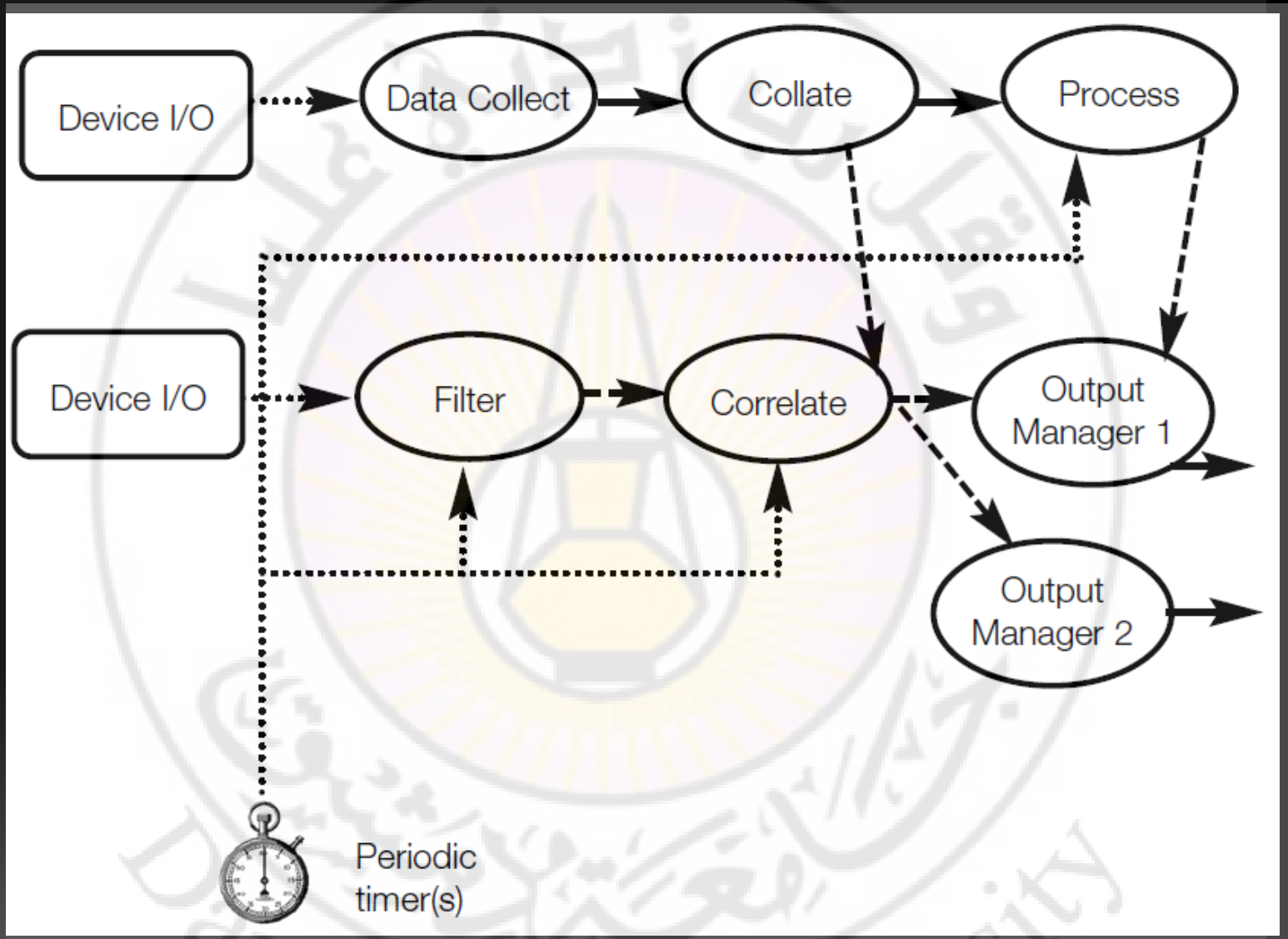
النظم الأنبوبية١
Pipelined Systems

- تستخدم النظم الأنبوبية رسائل بين المهام (غالبا ما تكون لها أولوية) بالإضافة إلى أجهزة إدخال/إخراج و مؤقتات من أجل إطلاق المهام
- يسري تدفق التحكم للحدث من خلال النظام اعتباراً من المنشأ وحتى بلوغ الأهداف
- لهذا يمكن وصف هذه النظم كمجموعة أنابيب استدعاء للمهام

النظم الأنبوبية٢
Pipelined Systems

- تلعب الأولويات في هذه الحالة دوراً ثانوياً
  - إذا كان الأنبوب وحيد الاتجاه فإن منح أولويات تصاعدية للمهام سوف يقلل من حجم رتل الرسائل
  - إذا كان الأنبوب ثنائي الاتجاه عندها يكون من الأفضل منح الأولويات بحيث تكون متكافئة على طول الأنبوب

PIPELINED SYSTEMS



Pipelined System نظام أنبوبي

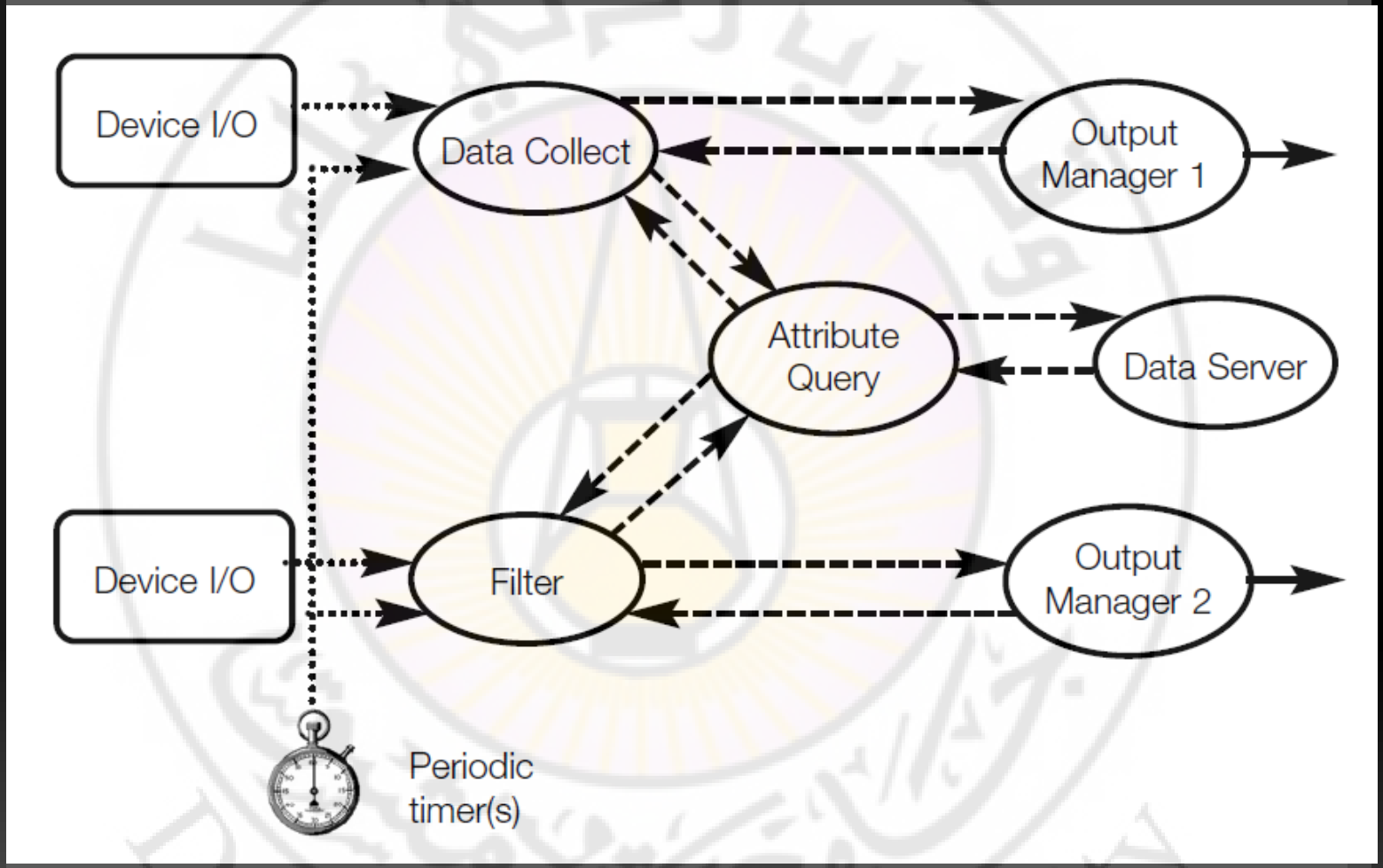
نظم مخدم/عميل١
Client-server systems

- تستخدم نظم مخدم/عميل رسائل بين المهام بالإضافة إلى أجهزة إدخال/إخراج و مؤقتات من أجل إطلاق المهام
- تدخل مهام الإرسال (عميل) في حالة تجميد block حتى استلام الجواب من مهام الاستقبال (مخدم)
- التحكم بالحدث يبقى على نظام واحد بينما يكون تدفق المعطيات موزعاً
- تكون العمليات الخاصة بمعالجة الأخطاء و نقاط المراقبة والتصحيح أسهل بالمقارنة مع النظم الأنبوبية

نظم مخدم/عميل٢
Client-server systems

- بشكل مشابه للنظم الأنبوبية تلعب الأولويات دوراً ثانوياً
- في الحالة المثالية ترث inherit مهام المخدم الأولوليات من العملاء إلا أن هذا الأمر غير عملي
- بشكل عملي تعطى المهام المختلفة نفس الأولوية وتستخدم الرسائل ذات الأولوية من أجل تجنب الوصول إلى حالة عنق الزجاجة

CLIENT-SERVER SYSTEMS



Client-server system نظام مخدم/عميل

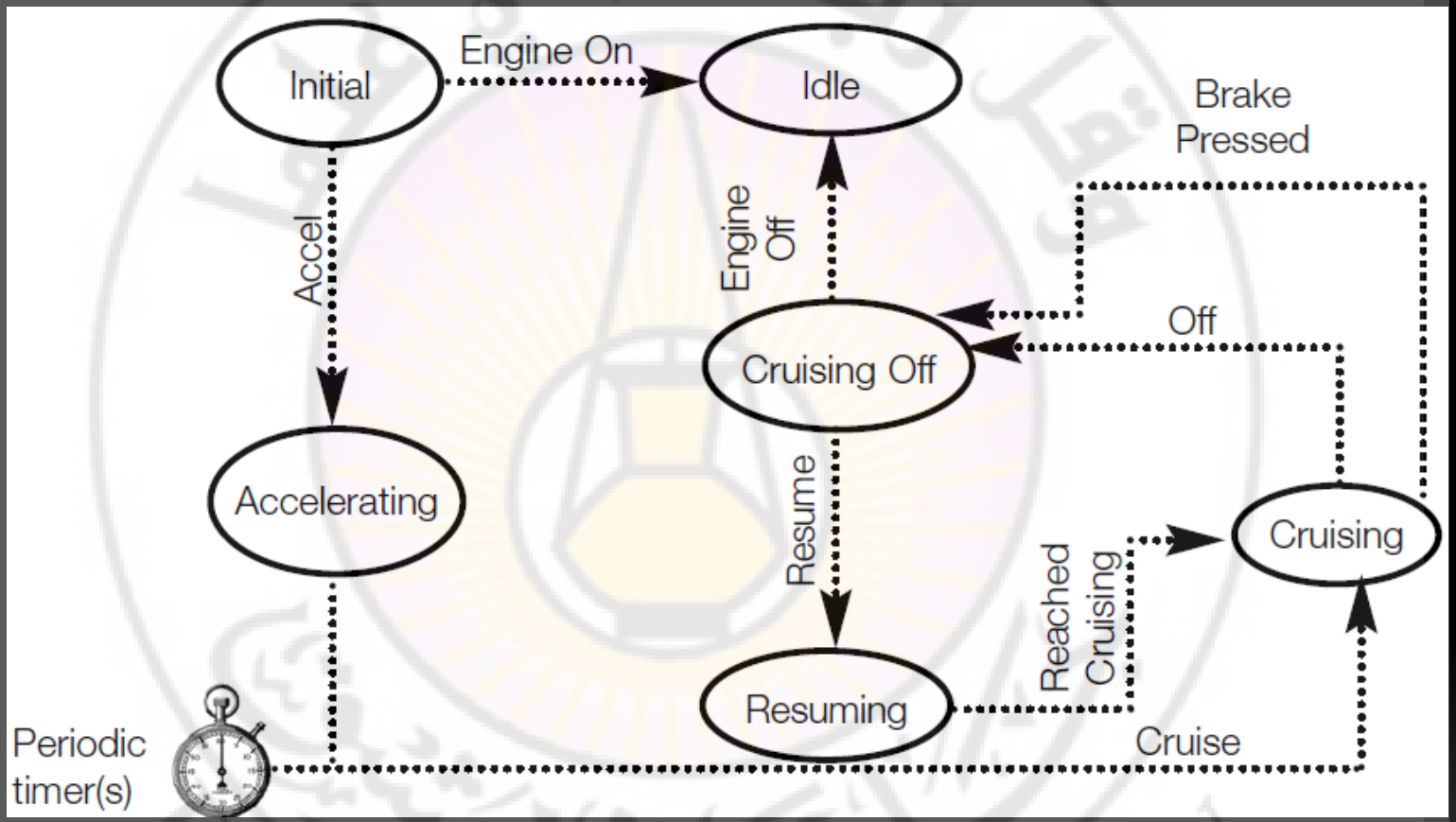
نظم آلة الحالة١
State machine systems

- في نظم آلة الحالة يتم تقسيم النظام إلى مجموعة من آلات الحالة النهائية الممتدة المتنافسة
- تستخدم كل آلة جزئية من أجل نمذجة سلوك الأغراض الفاعلة أو المتفاعلة
- تبقى الأغراض في إحدى الحالات المحددة بانتظار وقوع حدث
- وقوع حدث يمكن أن يؤدي إلى حدوث حالة انتقالية والتي تؤدي بدورها إلى تغير في الحالة بالإضافة إلى تنفيذ عمليات مرتبطة بانتقال الحالة

نظم آلة الحالة٢
State machine systems

- في نظم آلة الحالة يستمر التنفيذ حتى إتمام العمل حيث لا تقبل الآلة معالجة حدث حتى الانتهاء من الحدث السابق
- من أجل بناء نظام مكون من آلات حالة متنافسة يتم تحويل هذه الآلات إلى مجموعة من المهام المتنافسة
- من أجل ضمان خاصية التنفيذ حتى الإتمام يتم التحكم بآلة الحالة ضمن مسار واحد thread حيث يتم تنفيذ حلقة من أحداث الاستقبال والمعالجة

STATE MACHINE SYSTEMS



State machine system نظام آلة حالة

تقييم نماذج نظم الزمن الحقيقي ١

	إيجابيات	سلبيات
نظم التنفيذ الحلقي	•بسيطة •محددة deterministic •قابلة للتكرار •سهلة الفهم •الأكثر شيوعا •مناسبة لنظم الأمان الحرج safety-critical	•هشة fragile (إضافة، تعديل ..) •معقدة جداً عند الصيانة •مناسبة للنظم الصغيرة البسيطة التي لا تحتاج إمكانيات ديناميكية
النظم المقادة بالأحداث	•مقادة بالأولويات (مناسبة لـ RM) •بسيطة نسبياً •مناسبة للنظم القابلة للتحليل بشكل ستاتيكي (لا يوجد تغيير ديناميكي للأحمال)	غير قادرة على معالجة البيئات الموزعة بشكل فعال

تقييم نماذج نظم الزمن الحقيقي ٢

	إيجابيات	سلبيات
النظم الأنبوبية	•جاهزة للاستخدام في البيئات الموزعة (تعتمد كلياً على تبادل الرسائل)	•معقدة عند التحليل •أقل قابلية للتنبؤ من النظم الأخرى
نظم مخدم/عميل	•مناسبة للعمل ضمن النظم غرضية التوجه مثل CORBA •تتيح إمكانية تصحيح الأخطاء بسبب التغذية الراجعة ثنائية الاتجاه •مناسبة للعمل ضمن البيئات الموزعة	•معقدة عند التحليل •استهلاك إضافي للموارد بسبب حركة الرسائل الإضافية
آلة الحالة	•مناسبة للعمل ضمن النظم غرضية التوجه مثل CORBA	•معقدة عند التحليل

التحليل الزمني
Timing Analysis

- تعتبر عملية تقدير الاحتياجات الزمنية مهمة لأي تطبيق
- تصبح هذه الأهمية حاسمة بالنسبة لنظم الزمن الحقيقي
- تبدأ هذه العملية بتحديد القيود التي لا يمكن تجاوزها وتلك التي يمكن تجاوزها
- يجب ضمان احترام Hard deadline كأولولية عليا بغض النظر عن بقية القيود الزمنية

الغاية من التحليل الزمني١

- يقدم التحليل الزمني إطار من أجل جدولة الأحداث بحيث تكون الموارد الصلبة للنظام متاحة دائماً عند الحاجة من أجل ضمان احترام القيود الزمنية للمهام الحرجة
- الحد من المخاطر الزمنية: يعطي ضمانة من أجل احترام القيود الزمنية
- تخفيض كبير لزمن الاختبار باستخدام طرق تحليل مبرهنة تضمن بناء نظم زمن حقيقي يمكن التنبؤ بسلوكها بشكل أكيد

الغاية من التحليل الزمني٢

- ضمان التفاعل الصحيح بين مكونات النظام المتداخلة والتي تعمل بشكل متزامن ضمن أسوأ ظروف العمل المحتملة
- تحسين موثوقية النظام: ضمان عمل مكونات النظام ضمن المتوقع (تجنب الفشل) عند حدوث تأخير زمني غير اعتيادي لمهمة ضمن النظام
- اختبار مسبق: حتى قبل بناء النظام مما يخفض الكلفة والمخاطر الناجمة عن استخدام نمط أو عدد غير مناسب من المكونات

فوائد التحليل الزمني

- تحديد متطلبات النظام وتمريرها على شكل دفتر شروط لفريق التصميم والتطوير
- التمثيل المرئي لتكوين الأجزاء الصلبة والمرنة للنظام
- ضمان سلوك النظام بشكل قابل للتنبؤ
- تقدير سلوك النظام في الحالة الأسوأ worst-case
- تنفيذ تحليل what-if وتحديد التكوين الأفضل أو الأقل كلفة
- تجنب الأخطاء المكلفة
- ضمان بقاء موارد احتياطية في حالة النظم القابلة للتوسع

آليات الجدولة في الزمن الحقيقي١
Real-Time Scheduling Policies

- جدولة شفعية مع أولويات ثابتة:

Fixed Priority Preemptive Scheduling

- لكل مهمة task أولوية ثابتة لا تتغير (ما لم يقم التطبيق بتغييرها)
- تقوم المهام ذات الأولوية الأعلى بشفع المهام ذات الأولوية الأدنى
- معظم نظم الزمن الحقيقي تدعم هذه الآلية

آليات الجدولة في الزمن الحقيقي٢
Real-Time Scheduling Policies

- جدولة شفعية مع أولويات ديناميكية:

Dynamic-Priority Preemptive Scheduling

- يمكن للأولويات أن تتغير من نسخة Instance إلى أخرى أو حتى أثناء تنفيذ نسخة معينة من أجل تحقيق هدف ضمن زمن استجابة محدد
- تقوم المهام ذات الأولوية الأعلى بشفع المهام ذات الأولوية الأدنى
- القليل من نظم الزمن الحقيقي التجارية تدعم هذه الآلية

آليات الجدولة في الزمن الحقيقي٣
Real-Time Scheduling Policies

- جدولة المعدل الرتيب:

Rate-Monotonic Scheduling

- جدولة شفعية مع أولويات ثابتة حيث تزداد الأولوية بزيادة تواتر frequency المهمة
- تفترض هذه الآلية أن الحد الزمني deadline للمهمة هو عبارة عن دور المهمة نفسه
- يمكن تنفيذ هذه الآلية ضمن أي نظام يدعم الجدولة الشفعية مع أولويات ثابتة

آليات الجدولة في الزمن الحقيقي٤
Real-Time Scheduling Policies

- جدولة الحد الزمني الرتيب

Deadline-Monotonic Scheduling

- تعتبر بمثابة تعميم لجدولة المعدل الرتيب حيث يكون الحد الزمني للمهمة عبارة عن نقطة زمنية ثابتة نسبة إلى بداية المدة الزمنية للمهمة
- كلما كان الحد الزمني أقرب كلما ازدادت أولوية المهمة
- عندما يكون الحد الزمني مساوٍ للمدة الزمنية للمهمة تتحول هذه الآلية إلى ؟

آليات الجدولة في الزمن الحقيقي
Real-Time Scheduling Policies

- جدولة الحد الزمني الأقرب أولاً:

Earliest-Deadline-First Scheduling

- آلية جدولة شفعية مع أولويات ديناميكة
- الحد الزمني للمهمة في هذه الحالة هو اللحظة الزمنية المطلقة التي يتوجب إنهاء المهمة قبلها
- يتم حساب الحد الزمني عند خلق المهمة
- يقوم المجدول باختيار المهمة ذات الحد الزمني الأقرب أولاً
- تقوم المهمة ذات الحد الزمني الأقرب بشفع المهمة ذات الحد الزمني الأبعد
- تقوم هذه الآلية بتقليل التأخير الأعظمي لأي مجموعة من المهام بالمقارنة مع كل آليات الجدولة الأخرى

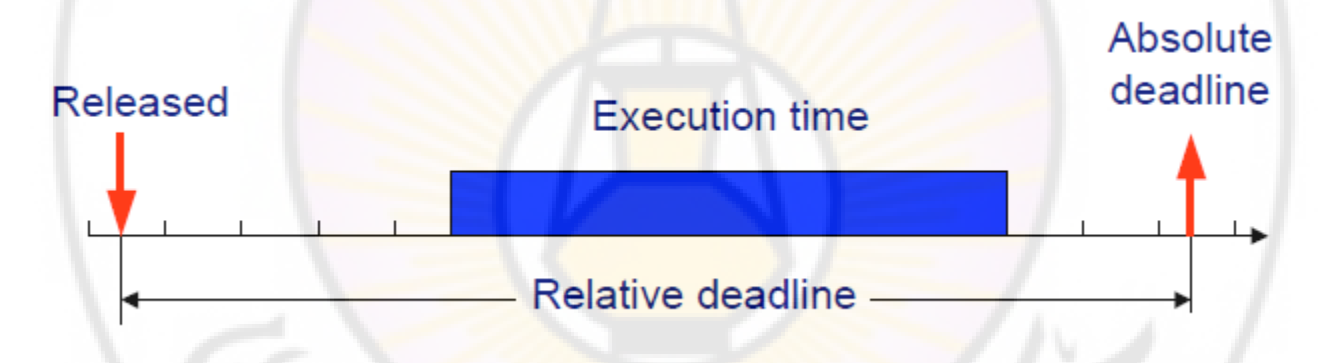
آليات الجدولة في الزمن الحقيقي٦
Real-Time Scheduling Policies

- جدولة التخامد الأدنى

Least Slack Scheduling

- جدولة غير شفعية مع أولويات ديناميكة
- تخامد المهمة هو عبارة عن حدها الزمني المطلق مع إنقاص زمن التنفيذ المتبقي (في الحالة الأسوأ) من أجل إنجاز المهمة
- يقوم المجدول باختيار المهمة ذات التخامد الأقصر للتنفيذ أولاً
- تقوم هذه الآلية بزيادة التأخيرات الدنيا لأي مجموعة من المهام

TIMING PARAMETERS

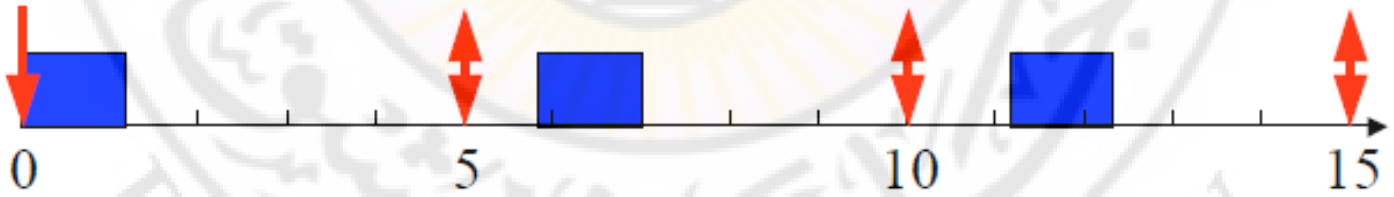


Timing parameters

PERIODIC TASK

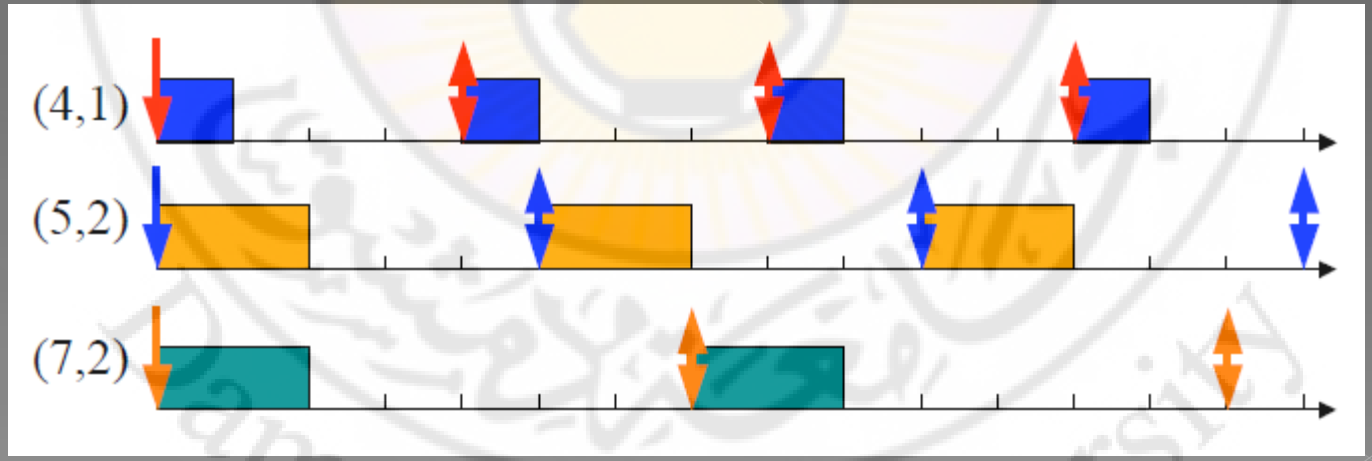
Task : a sequence of similar jobs

- Periodic task (p,e)
  - Its jobs repeat regularly
  - Period p = inter-release time ($0 < p$)
  - Execution time e = maximum execution time ($0 < e < p$)
  - Utilization U = e/p

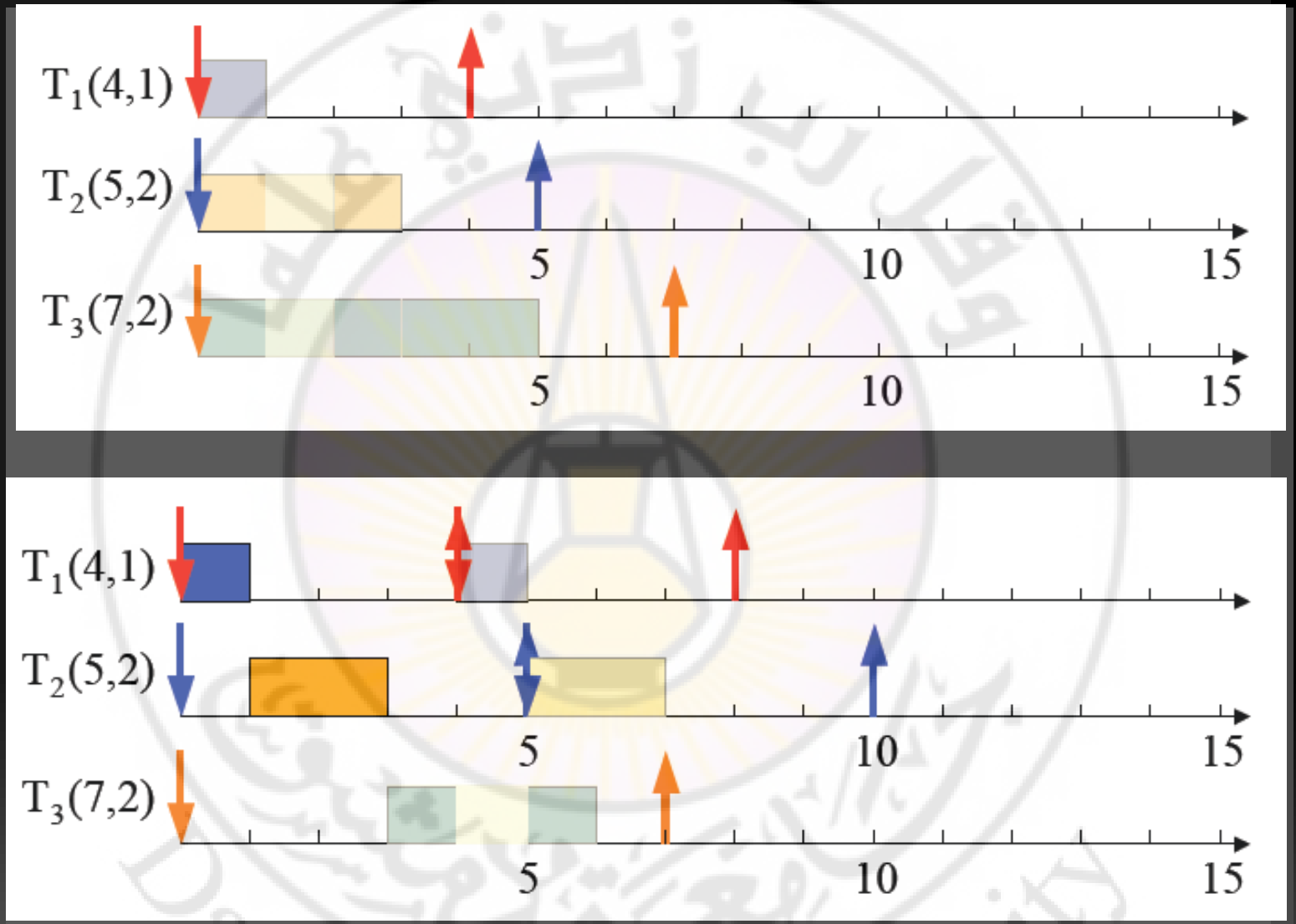


قابلية الجدولة
Schedulability

- خاصية تشير فيما إذا كان نظام زمن حقيقي (مجموعة مهام زمن حقيقي) قادر على احترام الحدود الزمنية المفروضة

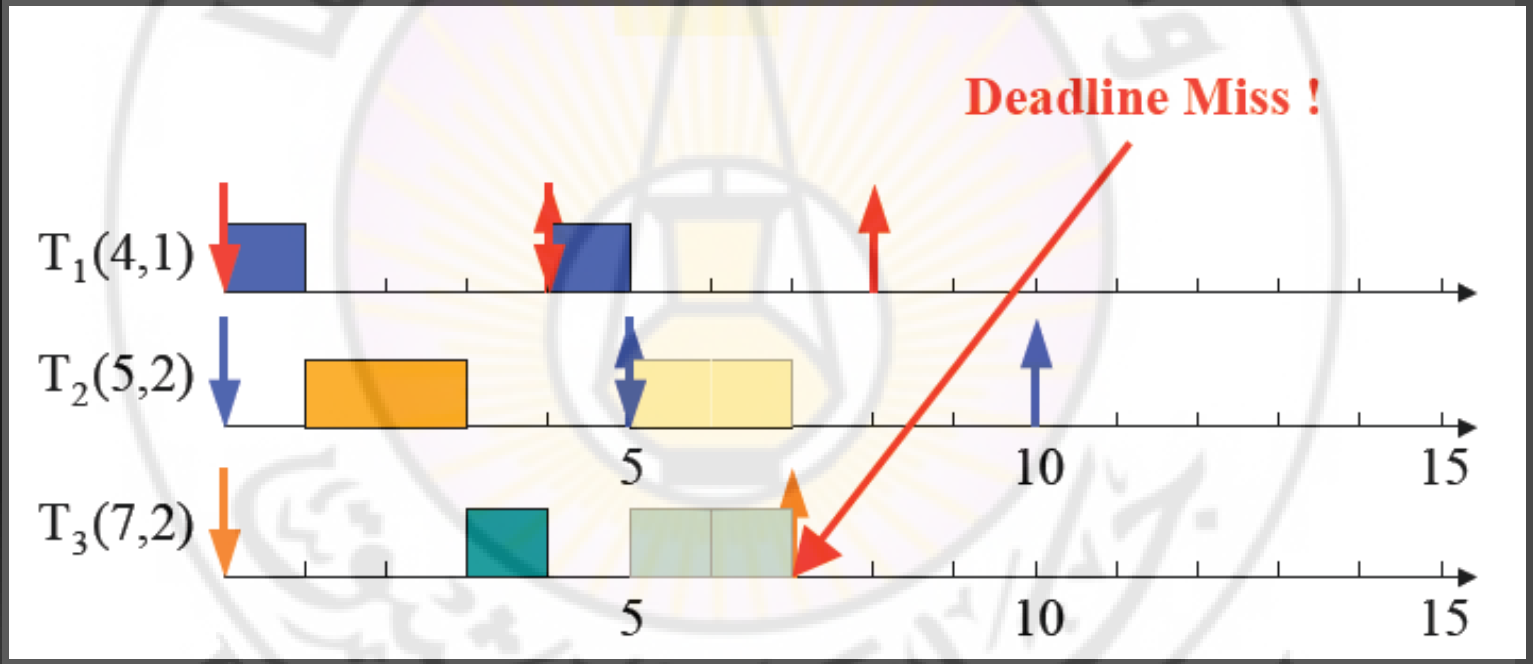


RM (RATE MONOTONIC)



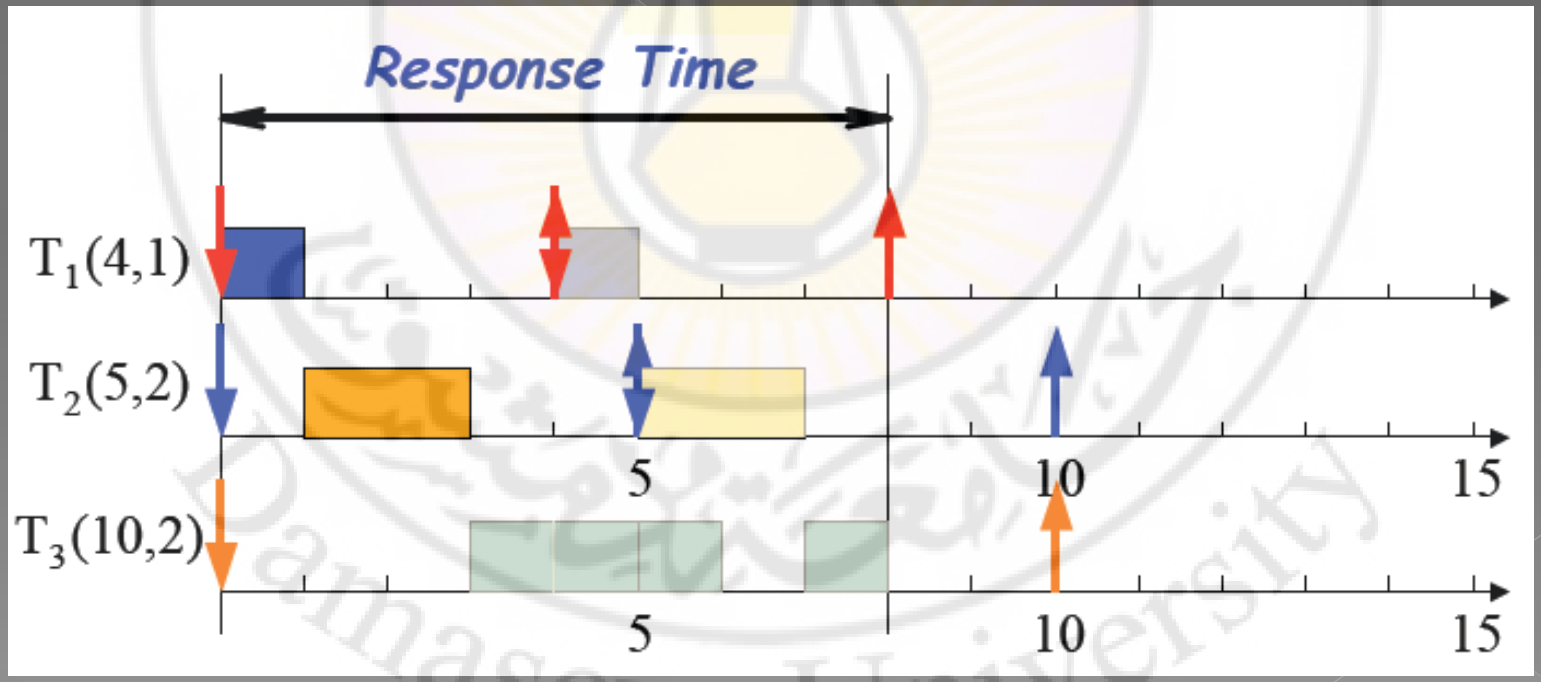
تنفيذ المهمة ذات الزمن الأقصر أولاً

RM (RATE MONOTONIC)



زمن الاستجابة
Response Time

- الفترة الزمنية الفاصلة بين لحظة إطلاق المهمة ولحظة إنهاءها

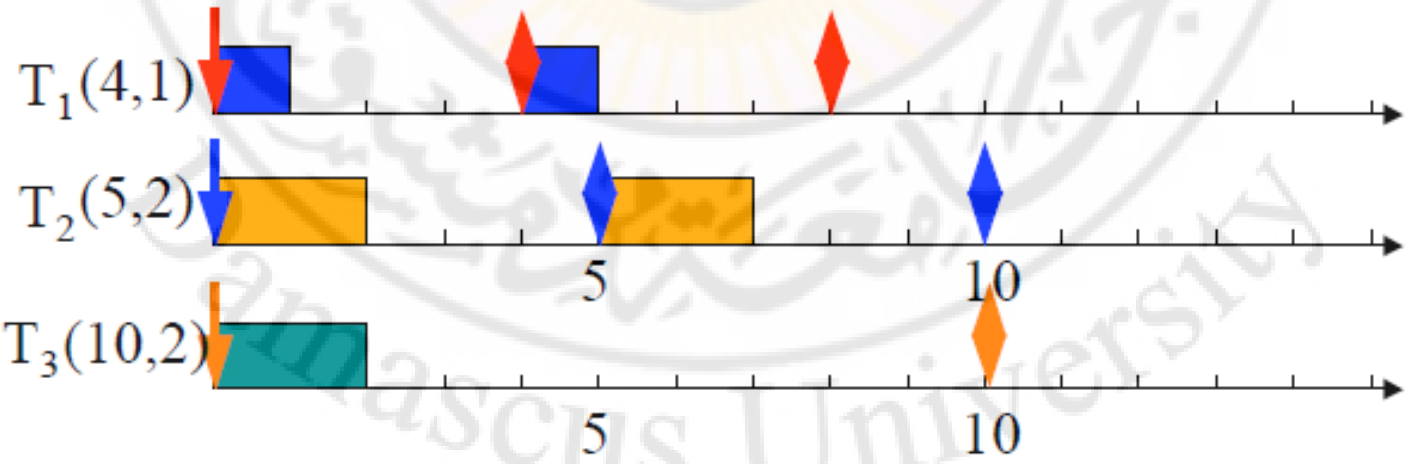


زمن الاستجابة
Response Time

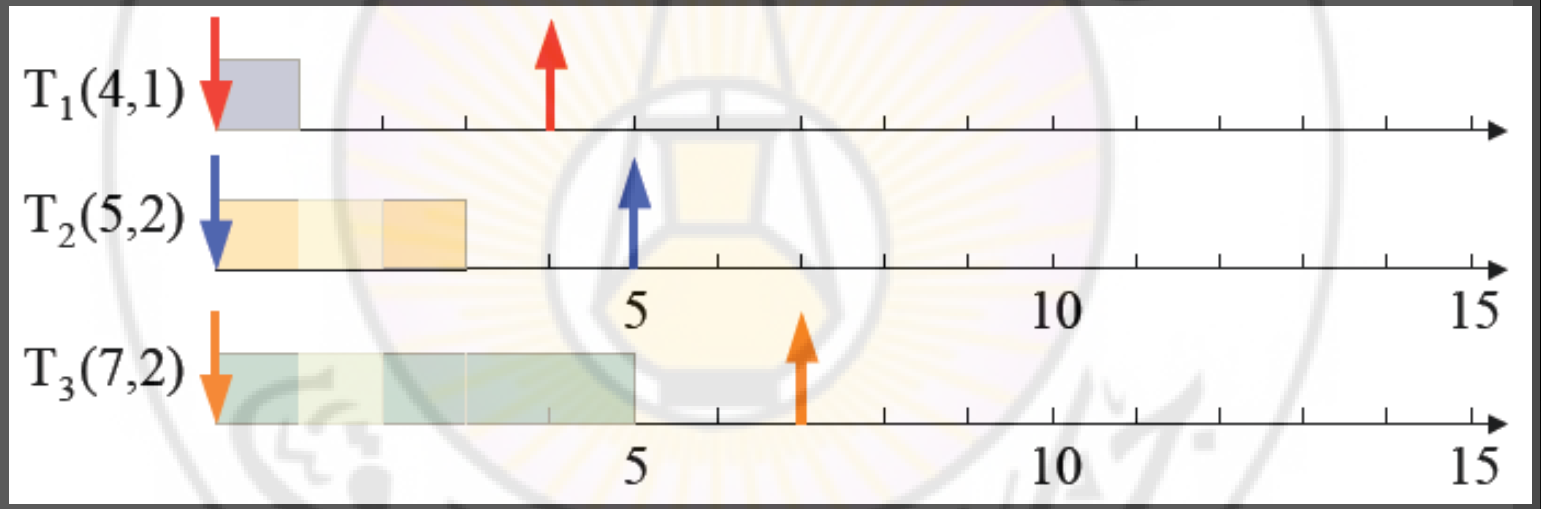
- Response Time (r_i)

$$r_i = e_i + \sum_{T_k \in HP(T_i)} \left\lceil \frac{r_i}{p_k} \right\rceil \cdot e_k$$

- HP(T_i) : a set of higher-priority tasks than T_i

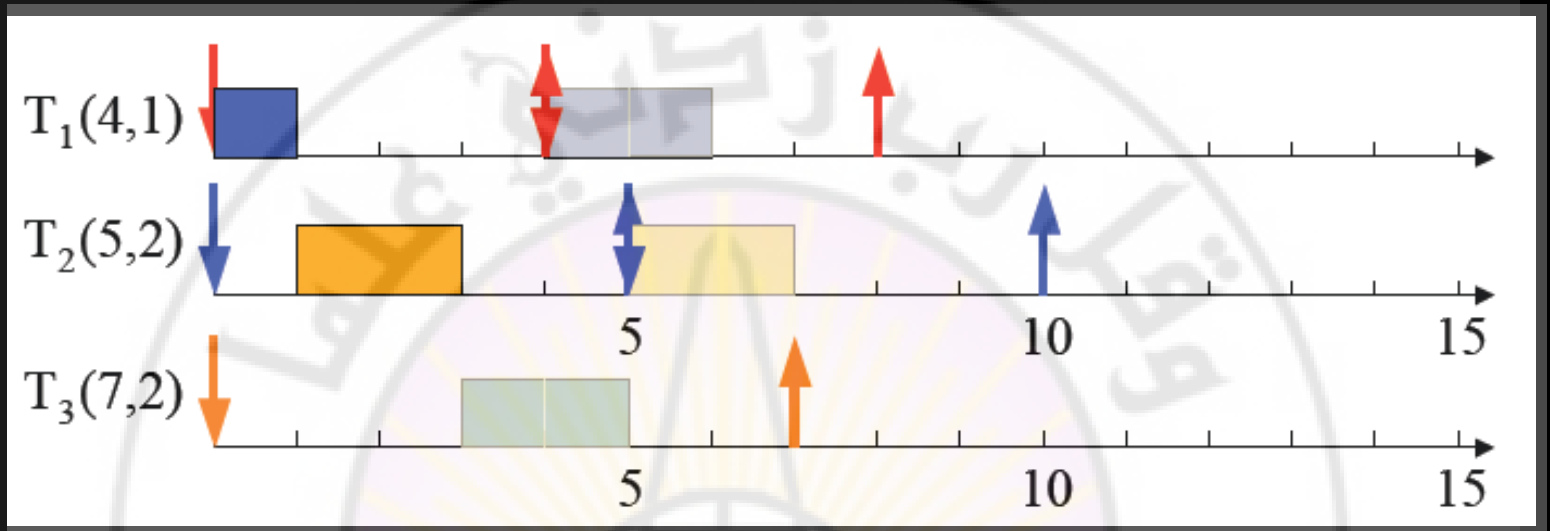


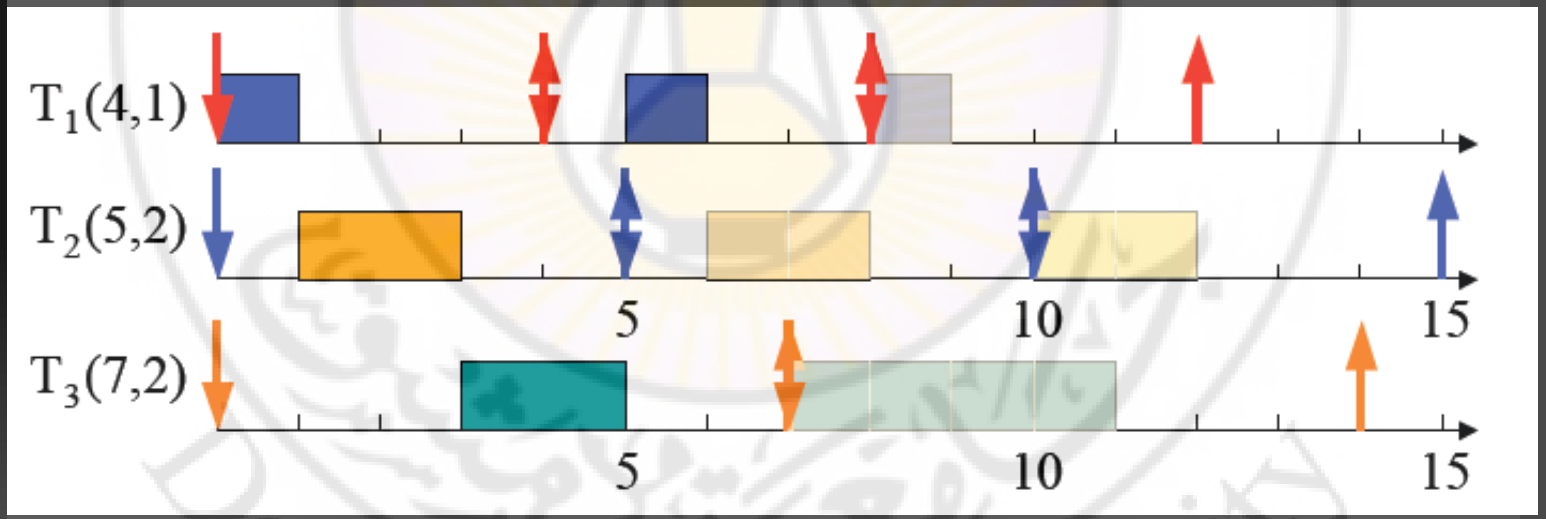
EDF (EARLIEST DEADLINE FIRST)



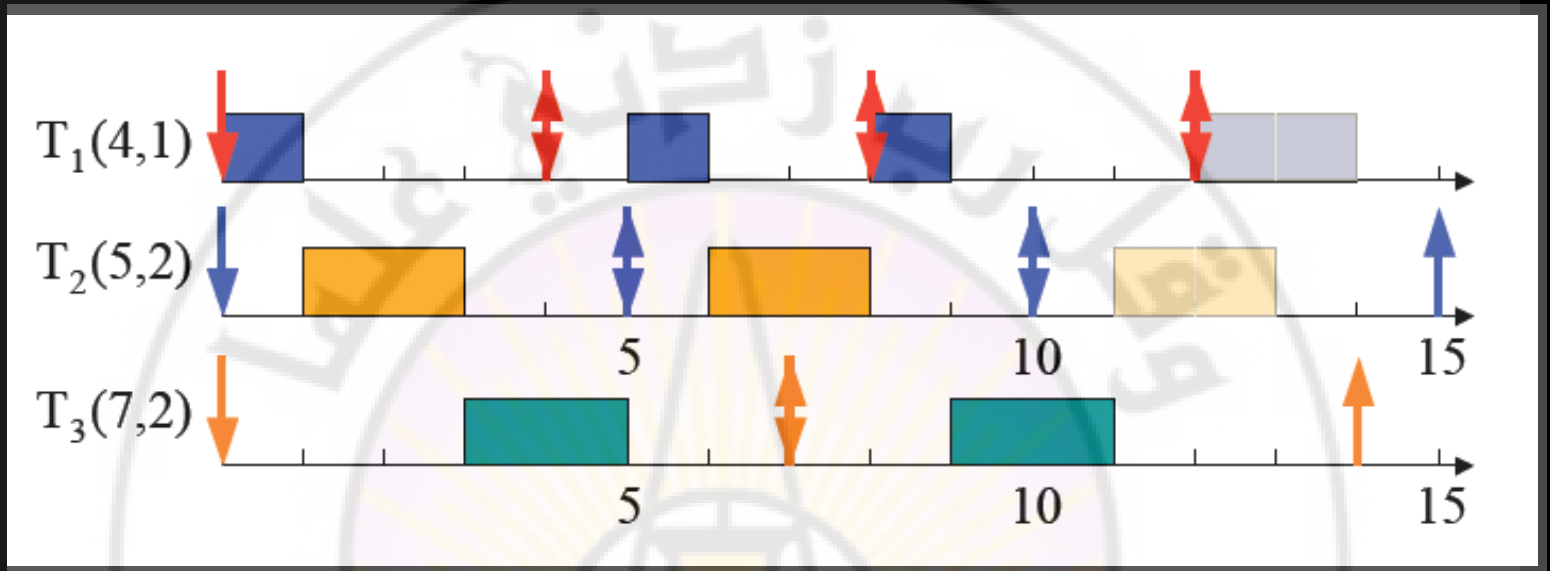
تنفيذ المهمة ذات الحد الزمني الأقرب

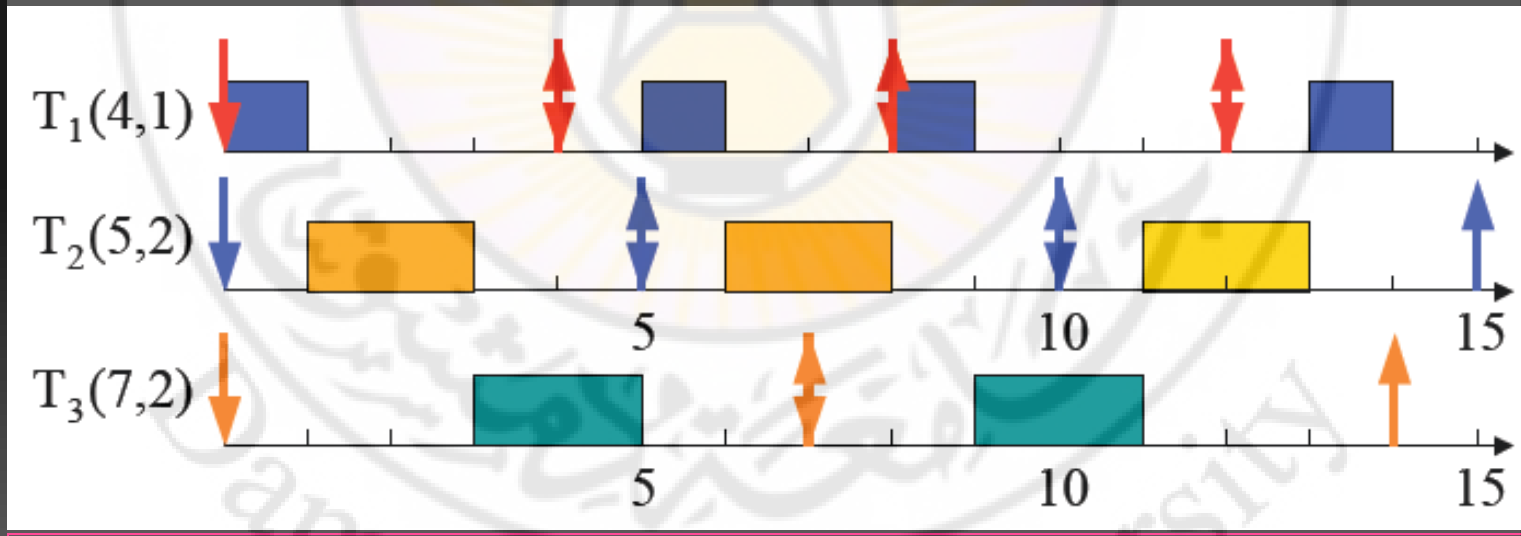
EDF (EARLIEST DEADLINE FIRST)





EDF (EARLIEST DEADLINE FIRST)





What is real-time scheduling theory (1)

- **Many real-time systems are built with operating systems providing multitasking facilities, in order to:**
  - Ease the design of complex systems (one function = one task).
  - Increase efficiency (I/O operations, multi-processor architecture).
  - Increase re-usability.

But, multitasking makes the predictability analysis difficult to do : we must take the task scheduling into account in order to check temporal constraints $\Longrightarrow$ schedulability analysis.

What is real-time scheduling theory (2)

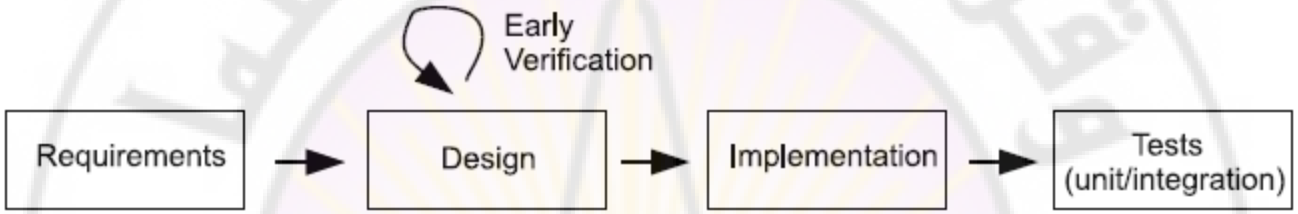
- **Example of a software embedded into a car:**

1. Tasks are released several times and have a job to do for each release.
2. Each task completes its current job before it has been released for the next one.
3. A task displays every 100 milliseconds the current speed of the car.
4. A task reads a speed sensor every 250 milliseconds.
5. A task performs an engine monitoring algorithm every 500 milliseconds.

$\Longrightarrow$ How can we check that tasks will be run at the proper rate? Do they meet their timing requirements? Do we have enough processor resource?

$\Longrightarrow$ If the system is complex (e.g. large number of tasks), the designer must be helped to perform such an analysis.

What is real-time scheduling theory (3)



- The real-time scheduling theory is a framework which provides:

1. **Algorithms to share a processor** (or any resources) by a set of tasks (or any resource users) according to some timing requirements $\Longrightarrow$ take urgency of the tasks into account.
2. **Analytical methods,** called **feasibility tests** or **schedulability tests**, which allow a system designer to early analyze/"compute" the system behavior before implementation/execution time.

Scheduling algorithms (1)

- **Different kinds of real-time schedulers:**
  - **On-line/off-line scheduler:** the scheduling is computed before or at execution time?
  - **Static/dynamic priority scheduler:** priorities may change at execution time?
  - **Preemptive or non preemptive scheduler:** can we stop a task during its execution ?
    1. Preemptive schedulers are more efficient than non preemptive schedulers (e.g. missed deadlines).
    2. Non preemptive schedulers ease the sharing of resources.
    3. Overhead due to context switches.

Scheduling algorithms (2)

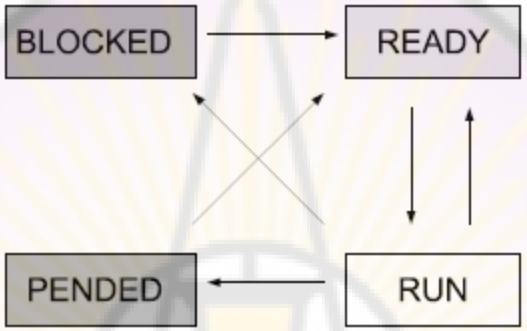
- **Different kinds of real-time schedulers:**
  - **Feasibility tests (schedulability tests) exist or not:** can we prove that tasks will meet deadlines before execution time?
  - **Complexity:** can we apply feasibility tests on large systems (e.g. large number of tasks)?
  - **Suitability:** can we implement the chosen scheduler in a real-time operating system?

Scheduling algorithms (3)

- What we look for when we choose a scheduler: we can compare them according to their ability to meet task deadlines.

  - A valid schedule is a schedule in which all task deadlines are met.
  - A feasible task set is a task set for which a valid schedule can be find.
  - **Optimality:** a scheduler a is optimal if it is able to find a valid schedule, each time a valid schedule exists for a task set.
  - **Dominant:** a is dominant comparing to b if all task sets that are feasible by scheduler b, are also feasible by a and if it exists some task sets that are feasible by a but not by b.
  - **Equivalent** : a and b are equivalent if all task sets that are feasible by a are also feasible by b and respectively.
  - **Incomparable** : a and b are incomparable is a can find a valid schedule for some task sets that b is not able to find and respectively.

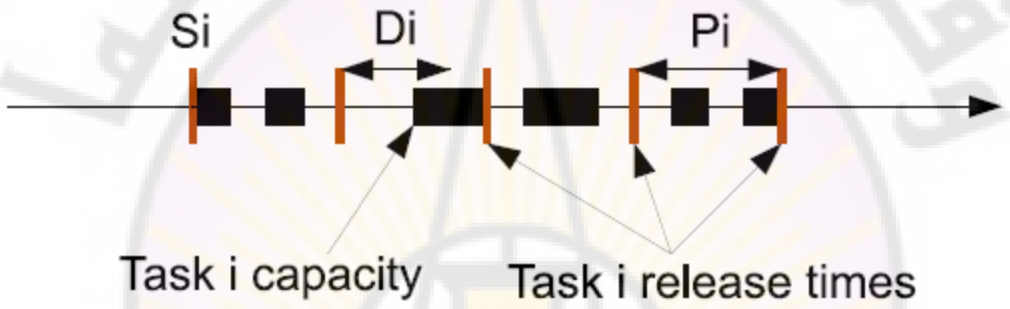
Models of task (1)



- **Task**: sequence of statements + data + execution context (processor and MMU). Usual states of a task.

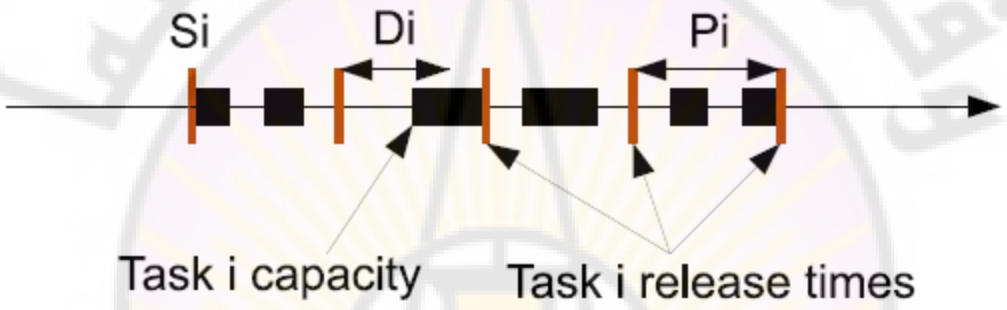
- **Usual task types**:
  - Urgent or/and critical tasks.
  - Independent tasks or dependent tasks.
  - Periodic and sporadic tasks (critical tasks). Aperiodic tasks (non critical tasks).

Models of task (2)



- **Usual parameters of a periodic task i:**
  - Period: P_i (duration between two periodic release times). A task starts a job for each release time.
  - Static deadline to meet: D_i, timing constraint relative to the period/job.
  - First task release time (first job): S_i.
  - Worst case execution time of each job: C_i (or capacity).
  - Priority: allows the scheduler to choose the task to run.

Models of task (3)



- **Assumptions for this lecture (synchronous periodic task with deadlines on requests) [LIU 73]:**

1. All tasks are periodic.
2. All tasks are independent.
3. $\forall i : P_i = D_i$: a task must end its current job before its next release time.
4. $\forall i : S_i = 0 \implies$ called critical instant (worst case on processor demand).

Usual real-time schedulers

1. **Fixed priority scheduler:** Rate Monotonic priority assignment (sometimes called Rate Monotonic Scheduling or Rate Monotonic Analysis, RM, RMS, RMA).
2. **Dynamic priority scheduler:** Earliest Deadline First (or EDF).

Verification of a real-time system

- How to perform verification of timing constraints of a real-time system (example):

1. Define/model hardware architecture and execution environment: capacity of the memory, processor, operating system features (and then the scheduler).
2. Implement functions (source code).
3. Design software architecture and software deployment on the hardware. Lead to a design of the software as a set of tasks, with their parameters and constraints.
4. Verify schedulability.
5. If task set is not feasible, go back to 1, 2 or 3.

Fixed priority scheduling (1)

- **Assumptions/properties of fixed priority scheduling :**
  - Scheduling based on fixed priority $\Longrightarrow$ static and critical applications.
  - Priorities are assigned at design time (off-line).
  - Efficient and simple feasibility tests.
  - Scheduler easy to implement into real-time operating systems.
- **Assumptions/properties of Rate Monotonic assignment:**
  - Optimal assignment in the case of fixed priority scheduling.
  - Periodic tasks only.

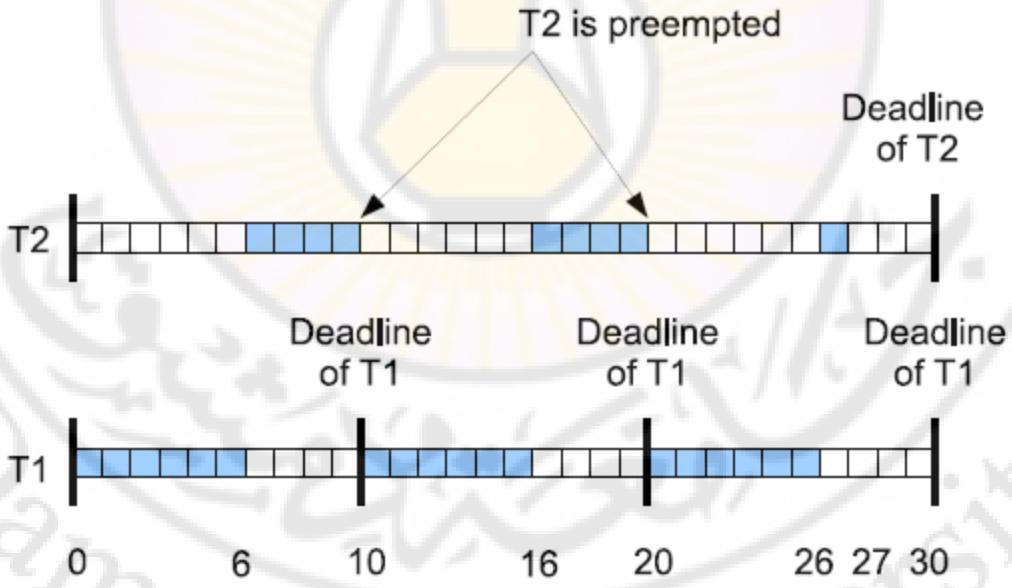
Fixed priority scheduling (2)

- **How does it work:**

1. **"Rate monotonic" task priority assignment:** the highest priority tasks have the smallest periods. Priorities are assigned off-line (e.g. at design time, before execution).
2. **Fixed priority scheduling:** at any time, run the ready task which has the highest priority level.

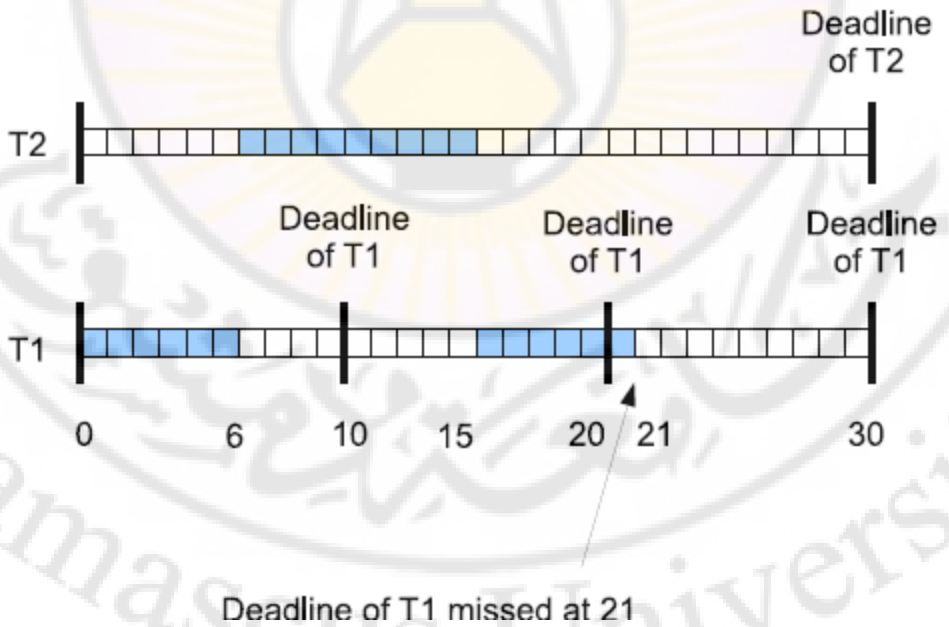
Fixed priority scheduling (3)

- Rate Monotonic assignment and preemptive scheduling:
  - Assuming VxWorks priority levels (high=0 ; low=255)
  - T1 : C1=6, P1=10, Prio1=0
  - T2 : C2=9, P2=30, Prio2=1



Fixed priority scheduling (4)

- Rate Monotonic assignment and non preemptive scheduling:
  - Assuming VxWorks priority levels (high=0 ; low=255)
  - T1 : C1=6, P1=10, Prio1=0
  - T2 : C2=9, P2=30, Prio2=1



Fixed priority scheduling (5)

- **Feasibility/Schedulability tests:**

1. **Run simulations on hyperperiod** = $[0, LCM(P_i)]$. Sufficient and necessary (exact result). Any priority assignment and preemptive/non preemptive scheduling.

2. **Processor utilization factor test**:

$$U = \sum_{i=1}^{n} \frac{C_i}{P_i} \leq n(2^{\frac{1}{n}} - 1)$$

   Rate Monotonic assignment and preemptive scheduling. Sufficient but not necessary. Does not compute an exact result.

3. **Task worst case response time, noted** $r_i \implies$ delay between task release time and task end time. Can compute an exact result. Any priority assignment and preemptive scheduling.

Fixed priority scheduling (6)

- **Compute r_i, the task worst case response time:**
  - Assumptions: preemptive scheduling, synchronous periodic tasks.
  - task i response time $=$ task i capacity + delay the task i has to wait due to higher priority task j. Or:

$$r_i = C_i + \sum_{j \in hp(i)} WaitingTime_j$$

or:

$$r_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{r_i}{P_j} \right\rceil C_j$$

  - $hp(i)$ is the set of tasks which have a higher priority than task i. $\lceil x \rceil$ returns the smallest integer not smaller than x.

Fixed priority scheduling (7)

- **To compute task response time**: compute w_i^k with:

$$w_i^{n+1} = C_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n}{P_j} \right\rceil C_j$$

- Start with $w_i^0 = C_i$.
- Compute w_i^1, w_i^2, w_i^3, ... w_i^k upto:
  - If $w_i^k > P_i$. No task response time can be computed for task i. Deadlines will be missed !
  - If $w_i^k = w_i^{k-1}$. w_i^k is the task i response time. Deadlines will be met.

Fixed priority scheduling (8)

- **Example:** T1 (P1=7, C1=3), T2 (P2=12, C2=2), T3 (P3=20, C3=5)

  - $w_1^0 = C1 = 3 \Longrightarrow r_1 = 3$
  - $w_2^0 = C2 = 2$
  - $w_2^1 = C2 + \left\lceil \frac{w_2^0}{P1} \right\rceil C1 = 2 + \lceil \frac{2}{7} \rceil 3 = 5$
  - $w_2^2 = C2 + \left\lceil \frac{w_2^1}{P1} \right\rceil C1 = 2 + \lceil \frac{5}{7} \rceil 3 = 5 \Longrightarrow r_2 = 5$
  - $w_3^0 = C3 = 5$
  - $w_3^1 = C3 + \left\lceil \frac{w_3^0}{P1} \right\rceil C1 + \left\lceil \frac{w_3^0}{P2} \right\rceil C2 = 5 + \lceil \frac{5}{7} \rceil 3 + \lceil \frac{5}{12} \rceil 2 = 10$
  - $w_3^2 = 5 + \lceil \frac{10}{7} \rceil 3 + \lceil \frac{10}{12} \rceil 2 = 13$
  - $w_3^3 = 5 + \lceil \frac{13}{7} \rceil 3 + \lceil \frac{13}{12} \rceil 2 = 15$
  - $w_3^4 = 5 + \lceil \frac{15}{7} \rceil 3 + \lceil \frac{15}{12} \rceil 2 = 18$
  - $w_3^5 = 5 + \lceil \frac{18}{7} \rceil 3 + \lceil \frac{18}{12} \rceil 2 = 18 \Longrightarrow r_3 = 18$

Fixed priority scheduling (9)

- How to schedule both periodic and aperiodic tasks, both critical and non critical functions:

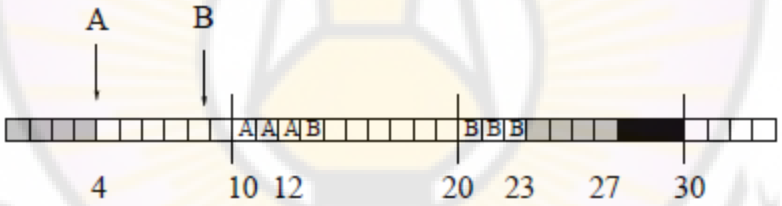
1. If aperiodic tasks are not urgent: give them a low priority level.
2. If aperiodic tasks are urgent: use aperiodic tasks servers.

Fixed priority scheduling (10)

- **Aperiodic tasks server**: periodic task devoted to the scheduling of aperiodic tasks.

T1 : C1=4 ; P1=20 (grey)
T2 : C2=12 ; P2=30 (white)
S : CS=4 ; PS=10;
A is released at t=4 ; CA=3
B is released at t=9 ; CB=4

Black=idle



- **Polling server:** run aperiodic task arrived before server activation. Does not use processor if no aperiodic task is present. Capacity is lost if no aperiodic task is present. Server stops aperiodic execution when its capacity is exhausted.
- **Many other servers:** sporadic server, priority exchange server, ...

Fixed priority scheduling (11)

- **Analysis of the car with embedded software example:**

1. $Tdisplay$: task which displays speed. P=100, C=20.
2. $Tspeed$: task which reads speed sensor. P=250, C=50.
3. $Tengine$: task which runs an engine monitoring program. P=500, C=150.

- **Processor utilization test:**

$U = \sum_{i=1}^{n} \frac{C_i}{P_i} = 20/100 + 150/500 + 50/250 = 0.7$

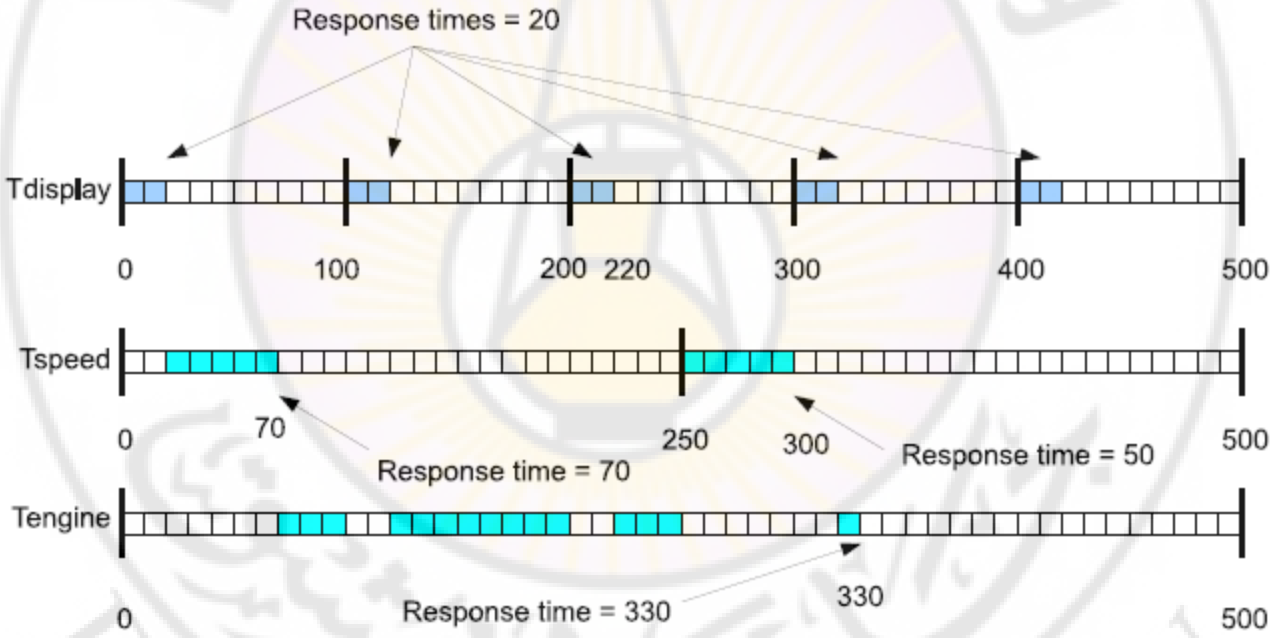
$Bound = n(2^{\frac{1}{n}} - 1) = 3(2^{\frac{1}{3}} - 1) = 0.779$

$U \leq Bound \Longrightarrow$ deadlines will be met.

- **Task response time:** $r_{Tengine} = 330$, $r_{Tdisplay} = 20$, $r_{Tspeed} = 70$.

Fixed priority scheduling (12)

- ... and check on the computed scheduling



- Run simulations on **scheduling period** = $[0, LCM(P_i)]$ = $[0, 500]$.

Earliest Deadline First (1)

- **Assumptions and properties:**
  - Dynamic priority scheduler $\Longrightarrow$ suitable for dynamic real-time systems.
  - Is able to schedule both aperiodic and periodic tasks.
  - Optimal scheduler: can reach 100 % of the cpu usage.
  - But difficult to implement into a real-time operating system.
  - Becomes unpredictable if the processor is over-loaded $\Longrightarrow$ not suitable for hard-critical real-time systems.

Earliest Deadline First (2)

- **How does it work:**

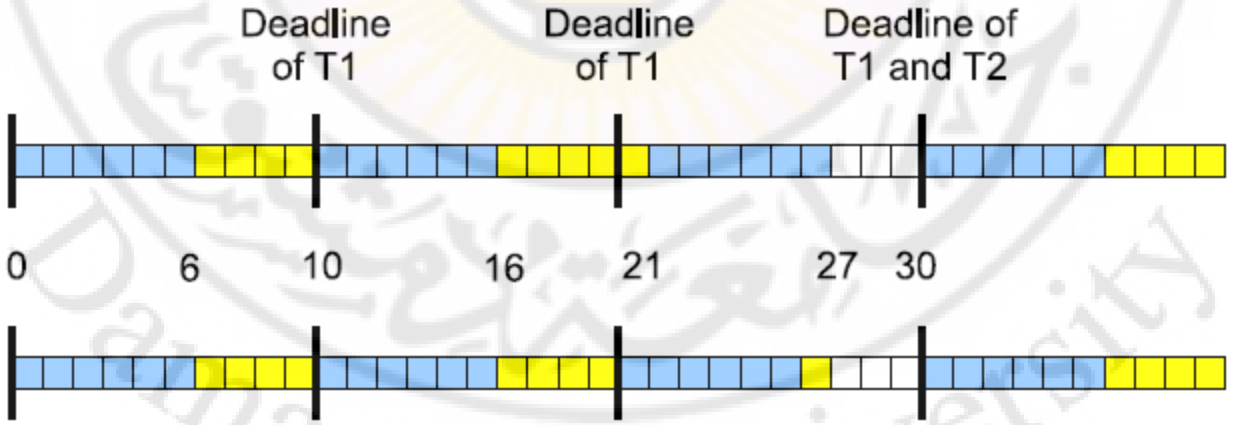
1. **Compute task priorities (called "dynamic deadlines")** $\Longrightarrow D_i(t)$ is the priority/dynamic deadline of task i at time t:
   - Aperiodic task : $D_i(t) = D_i + S_i$.
   - Periodic task: $D_i(t) = k + D_i$, where k is the task release time before t.

2. **Select the task:** at any time, run the ready task which has the shortest dynamic deadline.

Earliest Deadline First (3)

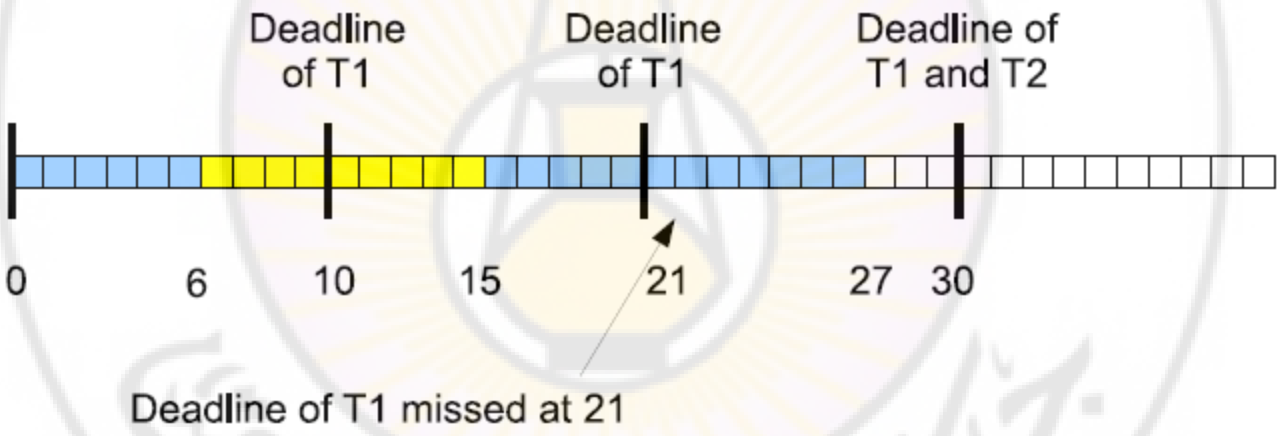
- Preemptive case: (T1/blue, T2/yellow, C1=6; P1=10; C2=9; P2=30)

t	$D_1(t)$	$D_2(t)$
0..9	$k + D_1 = 0 + 10 = 10$	$k + D_2 = 0 + 30 = 30$
10..19	$k + D_1 = 10 + 10 = 20$	30
20..29	$k + D_1 = 20 + 10 = 30$	30



Earliest Deadline First (4)

- Non preemptive case:



Earliest Deadline First (5)

- **Feasibility tests/schedulability tests:**

1. Run simulations on **hyperperiod** = $[0, LCM(P_i)]$. Sufficient and necessary (exact result).

2. **Processor utilization factor test** (e.g. preemptive case):

$$U = \sum_{i=1}^{n} \frac{C_i}{P_i} \leq 1$$

   Sufficient and necessary. Difficult to use in real life applications. Compute an exact result.

3. **Task response time:** a bit more complex to compute (dynamic scheduler) !

Multiprocessor/distributed (1)

- **Distributed systems:** "*A distributed system is a set of autonomous processors that are connected by a network and which have software to coordinate them self or share resources." Coulouris et al. [COU 94].* No shared memory. Message passing communication. A clock for each processor.
- **Multiprocessors systems:** a multiprocessors system is a set of autonomous processors which have software to coordinate them self and share resources but share the same clock and the same main memory.

- **Two scheduling approaches:**

1. Global scheduling.
2. Partitioning.

Multiprocessor/distributed (3)

- **Global scheduling:**
  - Few theoretical results (feasibility tests) compared to mono-processor real-time scheduling.
  - We can expect optimal processor usage: busy processors, less preemptions ... but task migrations.
  - Difficult to apply to heterogeneous systems: task migration, hardware, operating systems.
  - Well suited for multiprocessors architectures.

Multiprocessor/distributed (4)

- **Partitioning:**
  - Theoretical foundations of mono-processor scheduling compliant with current system implementation.
  - Non optimal resource sharing: a processor may stay idle, even if a task waits for a processor elsewhere in the system.
  - Better reliability, deterministic behavior in case of failure: failures of a task may only imply failure on its processor.
  - End-to-end worst case response time verification is difficult to do (too pessimistic most of the time).
  - Partitioning is a NP-hard problem: Bin-packing.
  - Well suited for current synchronous distributed systems (e.g. aircraft, ARINC 653).

Global scheduling (1)

- **Types de processors:**
  - **Identical:** processors have the same computing capability and run task at the same rate.
  - **Uniform:** each processor is characterised by its own computing capacity: for a processor of computing capacity s and for a work of duration t, the processor allocates $s \cdot t$ units of time to run the work.
  - **Specialised:** we define an execution rate for every uplet $(r_{i,j}, i, j)$: the work i requires $r_{i,j}$ units of time on the processor j.
  - Identical $\subset$ Uniform $\subset$ Specialized.
  - Specialized and uniform = heterogeneous processors.
  - Identical = homogeneous processors.

Global scheduling (3)

- **When migrations occur:**
  - **No migration:** a task must always run on a given processor=>partitioning.
  - **Job level migration:** each job of a task can be run on any processor. But a job started on a given processor can not be moved on another.
  - **Task level migration:** a task can run at any time on any processor.
- **Priority assignment:**
  - Fixed priority assigned to tasks (e.g. RM).
  - Fixed priority assigned to jobs (e.g. EDF).

Global scheduling (4)

- **Two types of scheduling algorithms:**

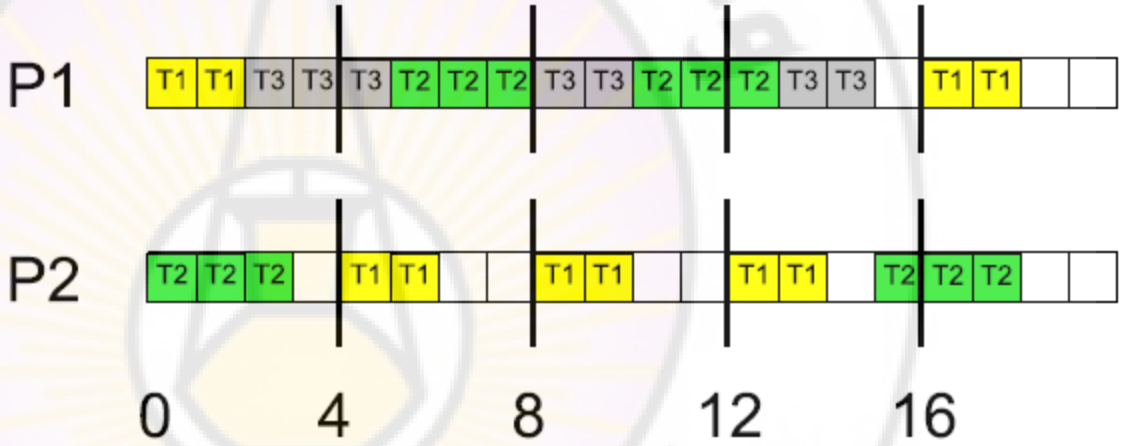
  1. Adapt mono-processor scheduling algorithms:
     - Global RM, global EDF, global DM, global LLF, ...
     - Choose the level of migration.
     - Apply the scheduling algorithm on all the set of the processors. At each time, assign m highest priority tasks to the m processors.
     - Preemptions occur when a job/task has to be run since it priority is high and when all processors are busy.
  2. New algorithms: PFair, LLREF, EDF(k), SA, EDZL, ...

- Be careful: multiprocessor scheduling is not a simple extension of monoprocessor scheduling $\Longrightarrow$ theoretical results are different and there are very few results.
- In the sequel, we assume identical processors.

Global scheduling (5)

- **Example:** global Deadline Monotonic

	Ci	Pi	Di
T1	2	4	4
T2	3	5	5
T3	7	20	20

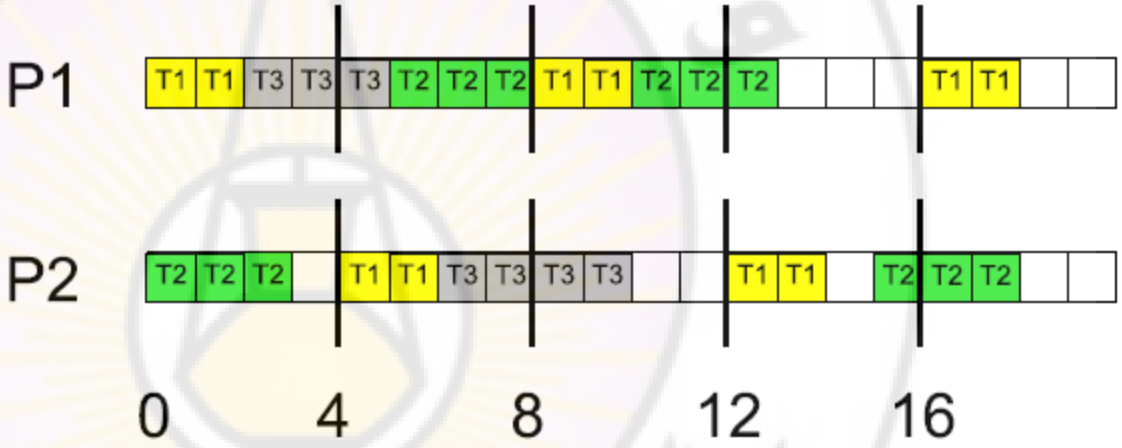


- Priority assignment: $T_1 > T_2 > T_3$.
- Job level migration.

Global scheduling (6)

- **Example:** global Deadline Monotonic

	Ci	Pi	Di
T1	2	4	4
T2	3	5	5
T3	7	20	20



- Priority assignment: $T_1 > T_2 > T_3$.
- Task level migration.

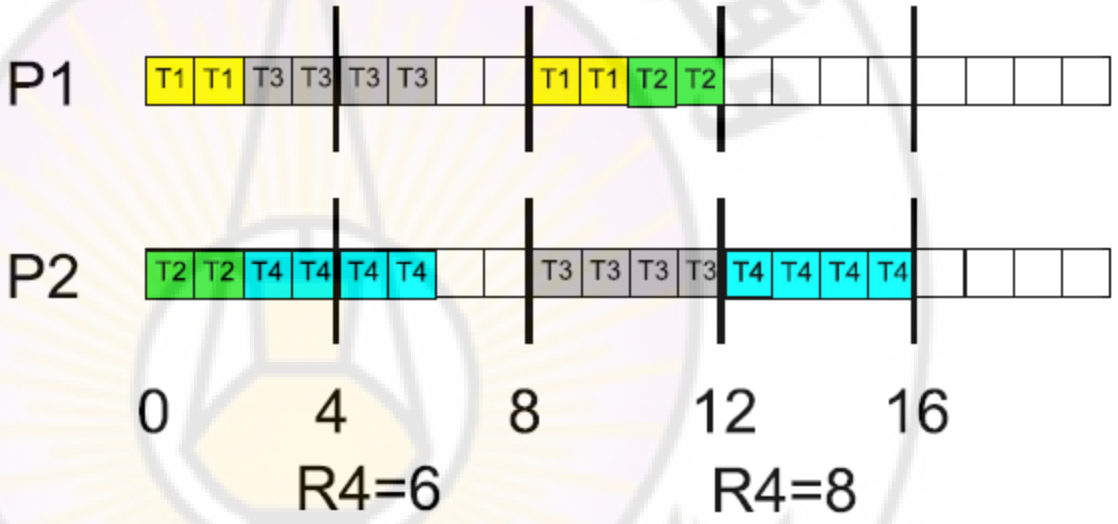
Global scheduling (7)

Task level migration	(1,3)	(2,3)	(3,3)
Job level migration	(1,2)	(2,2)	(3,2)
No migration	(1,1)	(2,1)	(3,1)
	1: task fixed priority	2: job fixed priority	3: dynamic priority at each unit of time

- **Comparing algorithms:**
  - Dynamic priority scheduling (3,3) are dominant. Optimality hierarchy is different from mono-processor hierarchy: global LLF is dominant comparing to global EDF.
  - (1,*) are incomparable each others.
  - (*,1) are incomparable with (*,2) and (*,3).
  - Pfair scheduler: identical processors + synchronous periodic tasks with deadline on request = optimal scheduler.

Global scheduling (8)

	Ci	Di	Pi
T1	2	2	8
T2	2	4	10
T3	4	6	8
T4	4	7	8



- **Critical instant:**
  - In the monoprocessor case with periodic tasks, critical instant is the worst case on processor demand and occurs in the beginning of the scheduling (busy period).
  - Assumption to compute worst case response times.
  - Is not true any more in the multiprocessor case.

Global scheduling (9)

- **Hyperperiod:**
  - In the monoprocessor case, the hyperperiod allows the verification:
    1. Of a periodic task set with any S_i.
    2. With any deterministic scheduler.

$$[0, lcm(\forall i : P_i) + 2 \cdot max(\forall i : S_i)]$$

  - In the multiprocessor case, only one known result:

$$[0, lcm(\forall i : P_i)]$$

    only for synchronous periodic task with deadline on request and preemptif fixed priority scheduling :-(

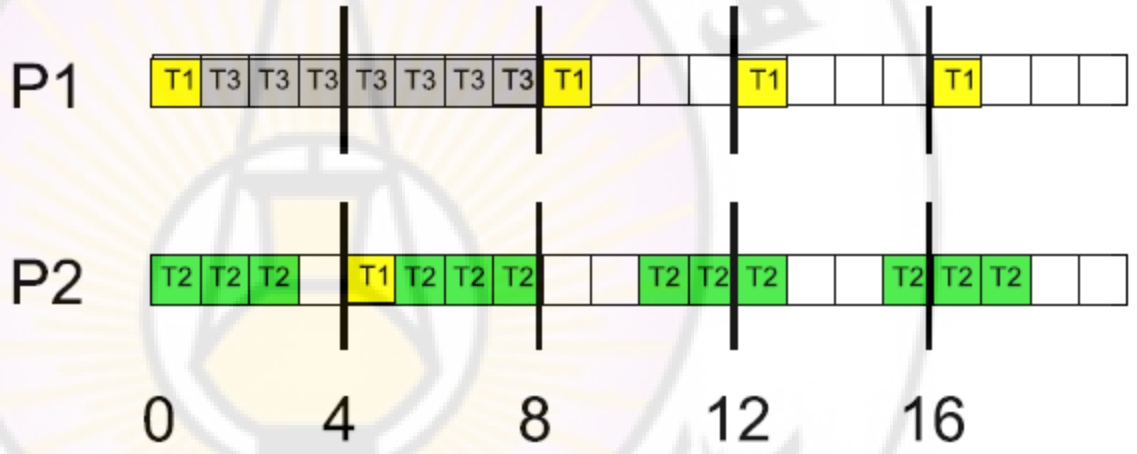
Global scheduling (10)

- **Scheduling errors:**
  - Errors: positive changes on a feasible task set lead to an unfeasible task set.
  - Scheduling verification is frequently made on worst cases. Example : a sporadic task set can be verified as a periodic task set.
  - Parameters that could be related to scheduling errors: C_i, $P_i \Longrightarrow$ decrease processor utilization factor.

Global scheduling (11)

- **Scheduling errors:**

	Ci	Pi	Di
T1	1	4	2
T2	3	5	3
T3	7	20	8

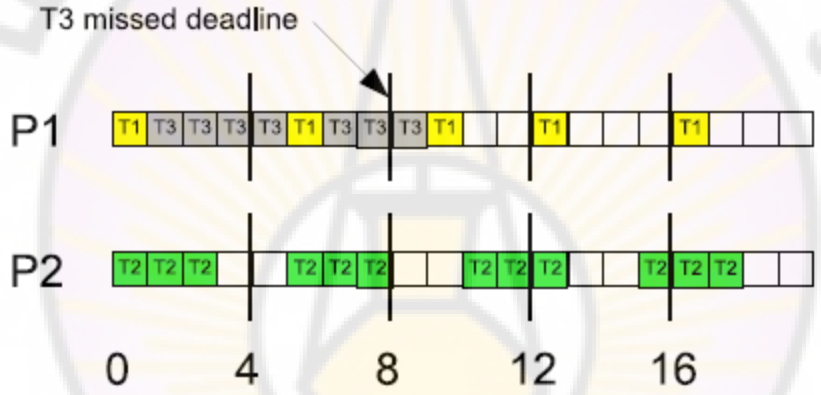


- Job migration level.
- Feasible task set.

Global scheduling (12)

- **Scheduling errors:**

	Ci	Pi	Di
T1	1	5	2
T2	3	5	3
T3	7	20	8



- Job level migration.
- Increasing P_1 ... leads to a task set which is not feasible any more.

Global scheduling (13)

- **Main ideas of the Pfair scheduling:**
  - We expect to find a schedule that leads to an equal processor utilization factor between all processors (called "Proportionate Fair" [AND 05]).
  - Optimal scheduler with identical processors and synchronous periodic tasks with deadline on request.
  - Deadline oriented.
  - Task level migrations.
  - Lead to many context switches.

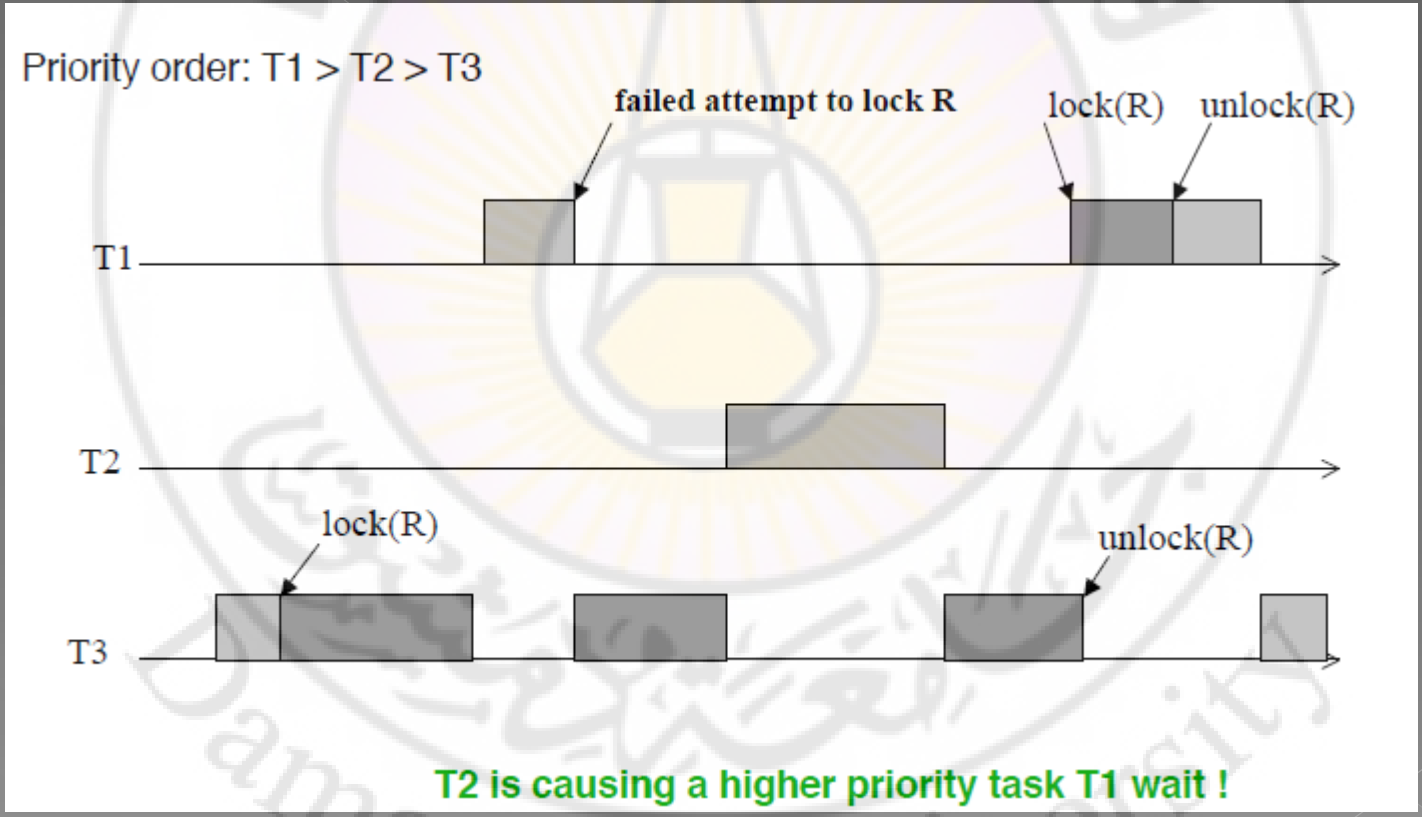
Priority Inversion and the MARS Pathfinder 1

- Landed on the Martian surface on July 4th, 1997
- Unconventional landing – boucing into the Martian surface
- A few days later, not long after Pathfinder started gathering meteorological data, the spacecraft began experiencing total system reset, each resulting in losses of data
- What happened:
  - Pathfinder has an "information bus"
  - The meteorological data gathering task ran as an infrequent, low priority thread, and used the information bus to publish its data (while holding the mutex on bus).

Priority Inversion and the MARS Pathfinder 2

- A communication task that ran with medium priority.
- It is possible for an interrupt to occur that caused (medium priority) communications task to be scheduled during the short interval of the (high priority) information bus thread was blocked waiting for the (low priority) meteorological data thread.
- After some time passed, a watch dog timer goes off, noticing that the data bus has not been executed for some time, it concluded that something had gone really bad, and initiated a total system reset.

The Priority Inversion Problem 1



The Priority Inversion Problem 2

- T1 has highest priority, T2 next, and T3 lowest
- T3 comes first, starts executing, and acquires some resource (say, a lock).
- T1 comes next, interrupts T3 as T1 has higher priority
- But T1 needs the resource locked by T3, so T1 gets blocked
- T3 resumes execution (this scenario is still acceptable so far)

The Priority Inversion Problem 3

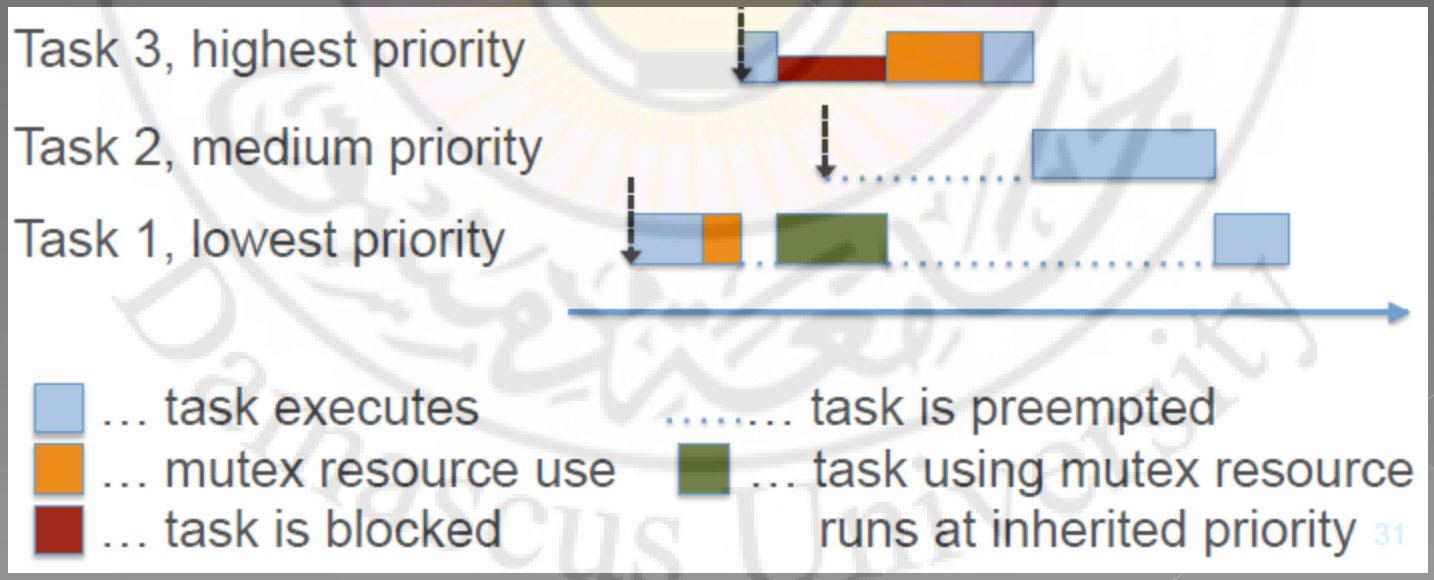
- T2 arrives, and interrupts T3 as T2 has higher priority than T3, and T2 executes till completion
- In effect, even though T1 has priority than T2, and arrived earlier than T2,T2 delayed execution of T1
- This is "priority inversion" !! Not acceptable.
- Solution T3 should inherit T1's priority at step 5

Priority Inversion

- Consider tasks with mutual exclusion constraints.
- Priority inversion is a phenomenon that occurs when a higher-priority task is blocked by a lower-priority task.
- Direct blocking: a high-priority task must not preempt the exclusive resource use by a low-priority task
- Indirect blocking of a high-priority task by a medium-priority task – the medium priority task preempts a low-priority task that holds a shared resource – has to be avoided.

Priority Inheritance 1

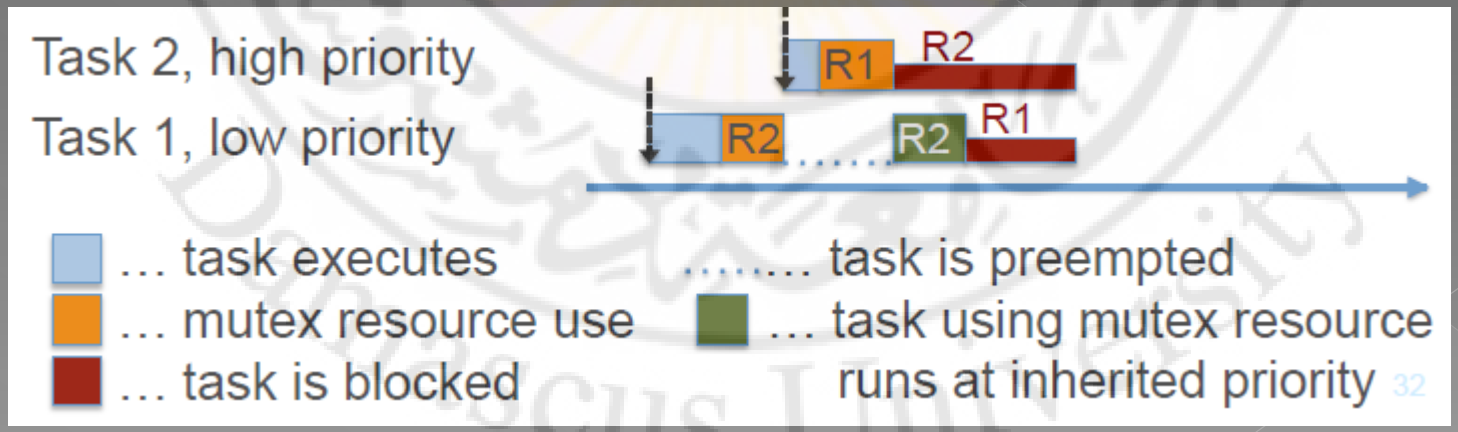
- When a low-priority task blocks one or more tasks of higher priority, it temporarily assumes the highest priority of a task it blocks



Priority Inheritance 2

- The priority-inheritance protocol does not prevent deadlocks
- Example

1. Task 1 locks R2
2. Task 2 preempts Task 1 and locks R1
3. Task 2 tries to lock R2 but fails
4. Task 1 inherits priority from Task 2 but blocks when trying to lock R1

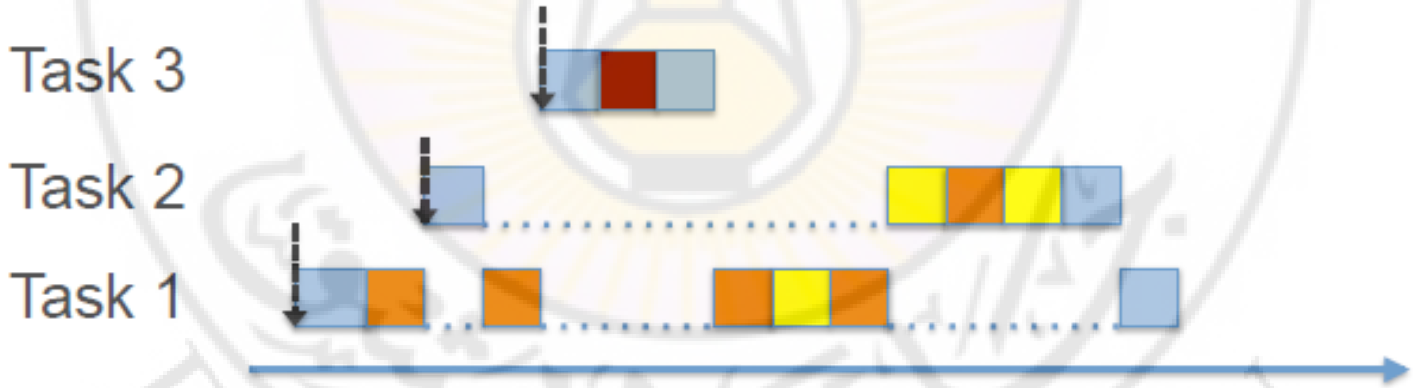


Priority Ceiling Protocol

- Each process has a default priority.
- Assign a priority ceiling to each resource:
- The priority ceiling equals the priority of the highest-priority task that uses the resource.
- At each time instant a task executes at a dynamic priority that is the maximum of its own static priority and the ceiling values of all recources that it has locked.
- ➪ A task can only assume a new resource if the task's priority is higher than the priority ceilings of all the resources locked by other tasks.

Priority Ceiling Protocol – Example

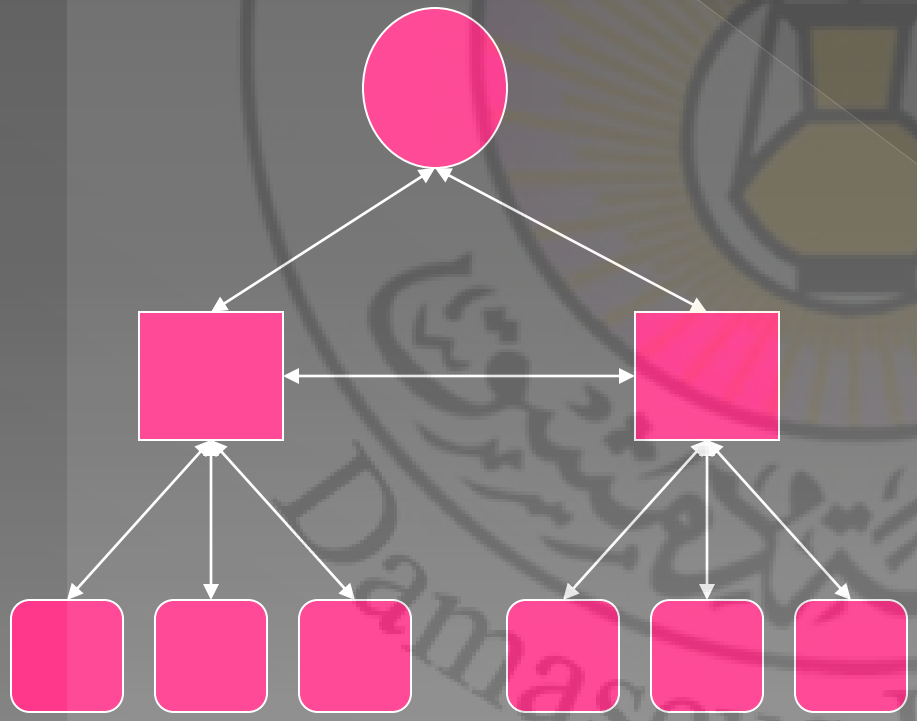
Task 3: … P(S1) … V(S1) …
Task 2: … P(S2) … P(S3) … V(S3) … V(S2) …
Task 1: … P(S3) … P(S2) … V(S2) … V(S3) …



Critical section guarded by Sx (priority ceiling):
… S1 (high) … S2 (medium) … S3 (medium)

Embedded and RTOS

Fire alarm system: an example



Central server

TCP/IP over radio

Controllers: ARM based

Low bandwidth radio links

Sensors: microcontroller based

Fire Alarm System

- Problem
  - Hundreds of sensors, each fitted with Low Range Wireless
    - Sensor information to be logged in a server & appropriate action initiated
- Possible Solution
  - Collaborative Action
    - Routing
      - Dynamic – Sensors/controllers may go down
      - Auto Configurable – No/easy human intervention.
      - Less Collision/Link Clogging
      - Less no of intermediate nodes
        - Fast Response Time
      - Secure

Role of an OS in Real Time Systems

- Standalone Applications
  - Often no OS involved
  - Micro controller based Embedded Systems
- Some Real Time Applications are huge & complex
  - Multiple threads
  - Complicated Synchronization Requirements
  - Filesystem / Network / Windowing support
  - OS primitives reduce the software design time

Features of RTOS's

- Scheduling.
- Resource Allocation.
- Interrupt Handling.
- Other issues like kernel size.

Scheduling in RTOS

- More information about the tasks are known
  - No of tasks
  - Resource Requirements
  - Release Time
  - Execution time
  - Deadlines
- Being a more deterministic system better scheduling algorithms can be devised.

Scheduling Algorithms in RTOS

- Clock Driven Scheduling
- Weighted Round Robin Scheduling
- Priority Scheduling
  (Greedy / List / Event Driven)

Scheduling Algorithms in RTOS *(contd)*

- Clock Driven
  - All parameters about jobs (release time/ execution time/deadline) known in advance.
  - Schedule can be computed offline or at some regular time instances.
  - Minimal runtime overhead.
  - Not suitable for many applications.

Scheduling Algorithms in RTOS (*contd*)

- Weighted Round Robin
  - Jobs scheduled in FIFO manner
  - Time quantum given to jobs is proportional to it's weight
  - Example use : High speed switching network
    - QOS guarantee.
  - Not suitable for precedence constrained jobs.
    - Job A can run only after Job B. No point in giving time quantum to Job B before Job A.

Scheduling Algorithms in RTOS (*contd*)

- Priority Scheduling (Greedy/List/Event Driven)
  - Processor never left idle when there are ready tasks
  - Processor allocated to processes according to priorities
  - Priorities
    - static - at design time
    - Dynamic - at runtime

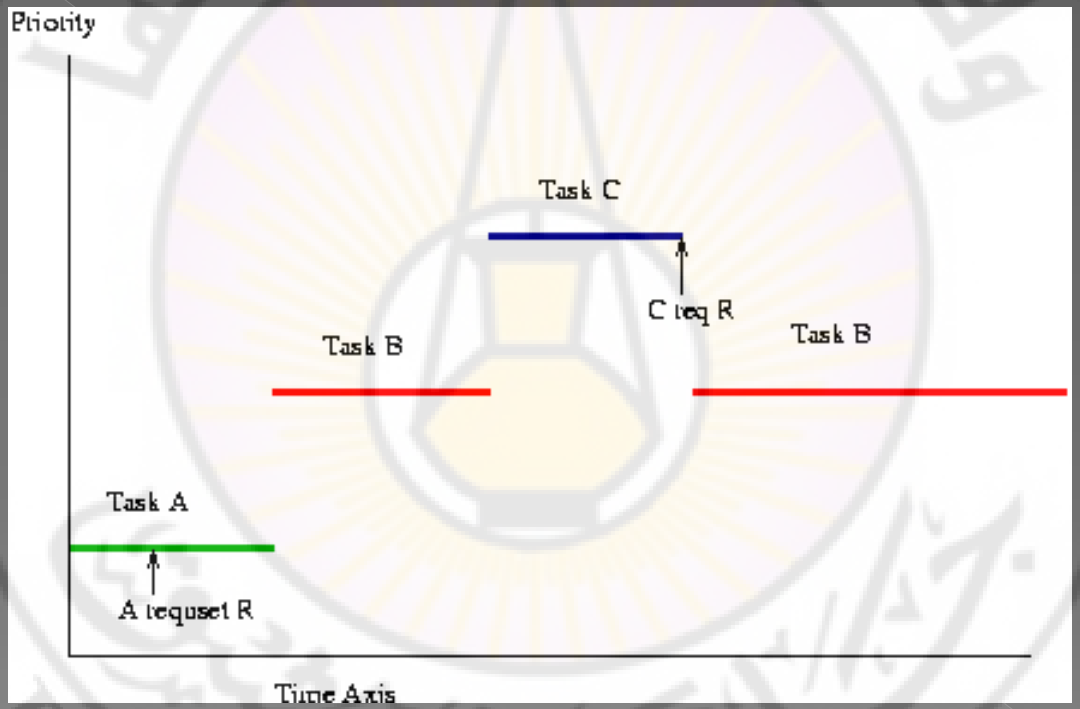
Priority Scheduling

- Earliest Deadline First (EDF)
  - Process with earliest deadline given highest priority
- Least Slack Time First (LSF)
  - slack = relative deadline – execution left
- Rate Monotonic Scheduling (RMS)
  - For periodic tasks
  - Tasks priority inversely proportional to it's period

Resource Allocation in RTOS

- Resource Allocation
  - The issues with scheduling applicable here.
  - Resources can be allocated in
    - Weighted Round Robin
    - Priority Based
- Some resources are non preemptible
  - Example : semaphores
- Priority Inversion if priority scheduling is used

Priority Inversion



Solutions to Priority Inversion

- Non Blocking Critical Section
  - Higher priority Thread may get blocked by unrelated low priority thread
- Priority Ceiling
  - Each resource has an assigned priority
  - Priority of thread is the highest of all priorities of the resources it's holding
- Priority Inheritance
  - The thread holding a resource inherits the priority of the thread blocked on that resource

Other RTOS issues

- Interrupt Latency should be very small
  - Kernel has to respond to real time events
  - Interrupts should be disabled for minimum possible time
- For embedded applications Kernel Size should be small
  - Should fit in ROM
- Sophisticated features can be removed
  - No Virtual Memory
  - No Protection

Linux for Real Time Applications

- Scheduling
  - Priority Driven Approach
    - Optimize average case response time
  - Interactive Processes Given Highest Priority
    - Aim to reduce response times of processes
  - Real Time Processes
    - Processes with high priority
    - No notion of deadlines
- Resource Allocation
  - No support for handling priority inversion

Interrupt Handling in Linux

- Interrupts are disabled in critical sections of the kernel
- No worst case bound on interrupt latency avaliable
  - eg: Disk Drivers may disable interrupt for few hundred milliseconds
- Not suitable for Real Time Applications
  - Interrupts may be missed

Other Problems with Linux

- Processes are non preemtible in Kernel Mode
  - System calls like fork take a lot of time
  - High priority thread might wait for a low priority thread to complete it's system call
- Processes are heavy weight
  - Context switch takes several hundred microseconds

Why Linux

- Coexistence of Real Time Applications with non Real Time Ones
  - Example http server
- Device Driver Base
- Stability

RTLinux

- Real Time Kernel at the lowest level
- Linux Kernel is a low priority thread
  - Executed only when no real time tasks
- Interrupts trapped by the Real Time Kernel and passed onto Linux Kernel
  - Software emulation to hardware interrupts
    - Interrupts are queued by RTLinux
    - Software emulation to disable_interrupt()

RTLinux (*contd*)

- Real Time Tasks
  - Statically allocate memory
  - No address space protection
- Non Real Time Tasks are developed in Linux
- Communication
  - Queues
  - Shared memory

RTLinux Framework