Chapter 4-1: Processes and Operating Systems

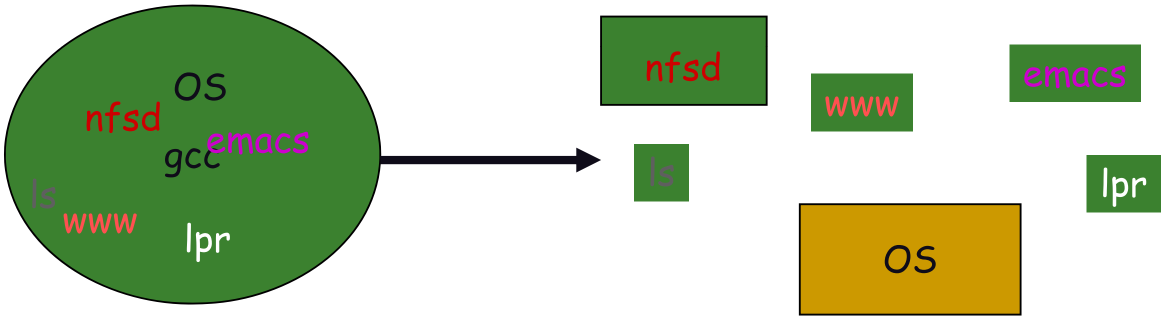
Soo-Ik Chae

Topics

- Processes and threads
- Real-time scheduling.

Why processes? Simplicity

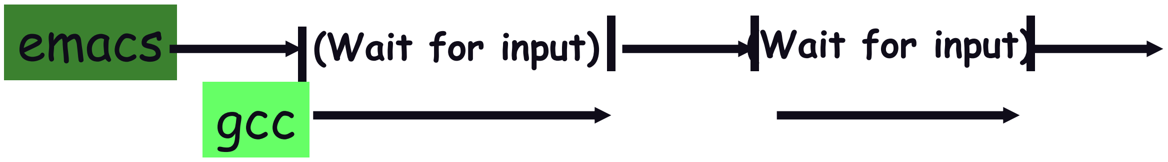
- Many things going on in system



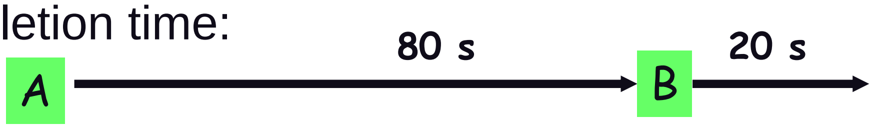
- How to make it simple?
  - Separate each in isolated process. OS deals with one thing at a time, they just deal with OS
  - The universal trick for managing complexity: decomposition (“reductionism”)

Why processes? Speed

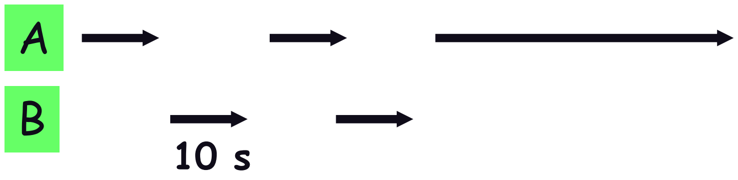
- I/O parallelism:



  - Overlap execution: make 1 CPU into many
  - (Real parallelism: > 1 CPU (multiprocessing))
- Completion time:



  - B’s completion time = 100s (A + B) So overlap



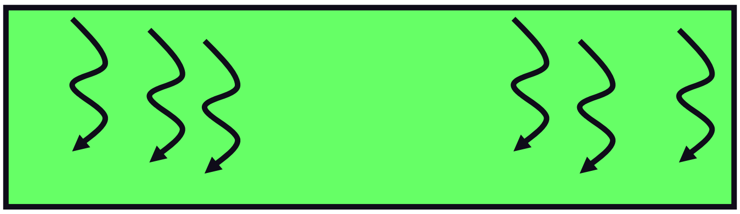
What is a thread?

- What's needed to run code on CPU
  - "execution stream in an execution context"
  - Execution stream: a sequence of instructions
- CPU execution context (1 thread)
  - State: stack, heap, registers
  - Position: program counter register

```
add r1, r2, r3
sub r2, r3, r10
 st r2, 0(r1)
        …
```

What is a process?

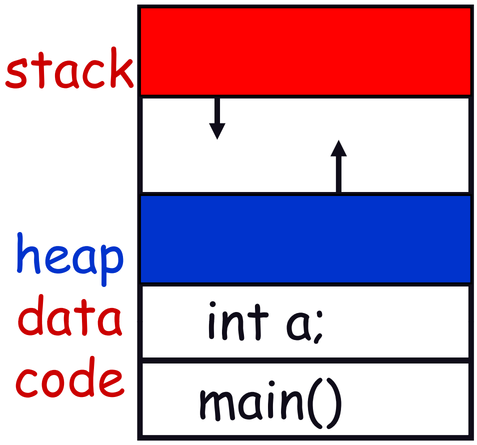
- Process: threads + address space
  - or, abstraction representing what you need to run thread on OS (open files, etc)
- Address space: encapsulates protection
  - address state passive, threads active
- Why separate thread, process?
  - Many situations where you want multiple threads per address space (servers, OS, parallel program)



Process != Program

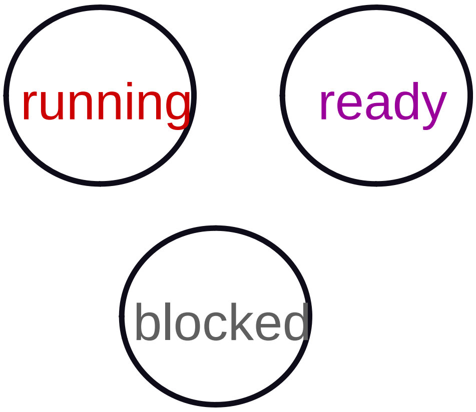
```
int c;
int main() {
  printf(“hello”);
}
```

- Program: code + data
  - passive
- Process: running program
  - state: registers, stack, heap…
  - position: Program counter



Process states

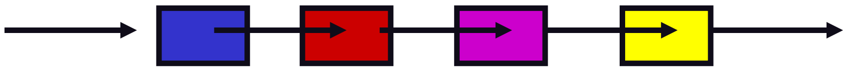
- Processes in three states:



  - Running: executing now
  - Ready: waiting for CPU
  - Blocked: waiting for another event (I/O, lock)
- Which ready process to pick?
  - 0 ready processes: run idle loop
  - 1 ready process: easy!
  - > 1: what to do?

Picking a process to run

- Scan process table for first runnable?
  - Expensive. Weird priorities (small pid's better)
  - Divide into runnable and blocked processes
- FIFO?
  - Put threads on back of list, pull them off from front



- Priority?
  - Give some threads a better shot at the CPU
  - problem?

Scheduling policies

- Scheduling issues
  - Fairness: don't starve process
  - Prioritize: more important first
  - Deadlines: must do by time 'x' (car brakes)
  - Optimization: some schedules >> faster than others
- No universal policy:
  - Many variables, can't maximize them all
  - Conflicting goals
    - **More important jobs vs starving others**
    - **I want my job to run first, you want yours.**
- Given some policy, how to get control? Switch?

Real-time scheduling terminology

- Process: unique execution of a program
- Context switch: operating system switch from one process to another.
- Time quantum: time between OS interrupts.
- Schedule: sequence of process executions or context switches.
- Thread: process that shares address space with other threads.
- Task: a collection of processes.
- Subtask: one process in a task.

Real-time scheduling algorithms

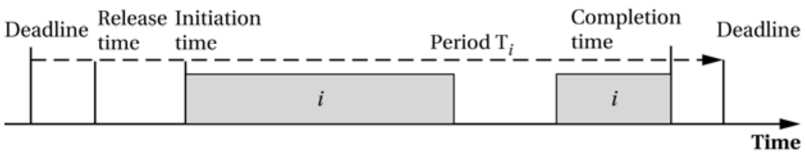
- Static scheduling algorithms determine the schedule off-line before the system begins to operate.
  - Constructive algorithms don’t have a complete schedule until the end of the scheduling algorithm.
  - Iterative improvement algorithms build a schedule, then modify it.
- Dynamic scheduling algorithms build the schedule during system operation.
  - Priority schedulers assign priorities to processes.
  - Priorities may be static or dynamic.

Timing requirements

- Real-time systems have timing requirements.
    - Hard: missing a deadline causes system failure.
    - Soft: missing a deadline does not cause failure.
- Deadline: time at which computation must finish.
- Release time: first time that computation may start.
- Period (T): interval between deadlines.
- Relative deadline: release time to deadline.

Timing behavior

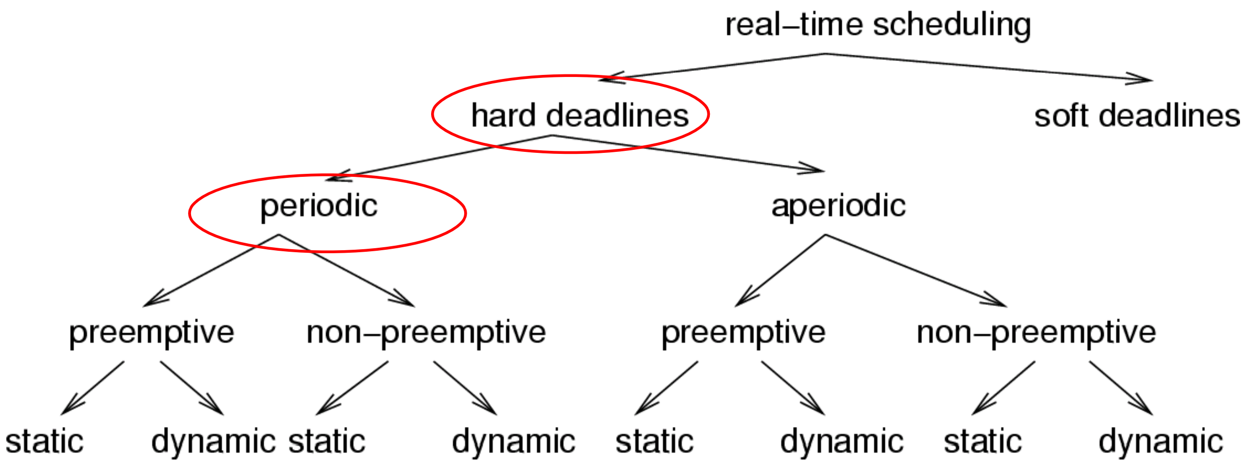
- Initiation time: time when process actually starts executing.
- Completion time: time when process finishes.
- Response time = completion time – release time.
- Execution time (C): amount of time required to run the process on the CPU.



Utilization

- Total execution time C required to execute processes 1..n is the sum of the C_is for the processes.
- Given available time t, utilization U = C/t.
  - Generally expressed as a percentage.
  - CPU can’t deliver more than 100% utilization.

Classification of scheduling algorithms



real–time scheduling

hard deadlines

soft deadlines

periodic

aperiodic

preemptive

non–preemptive

preemptive

non–preemptive

static

dynamic

static

dynamic

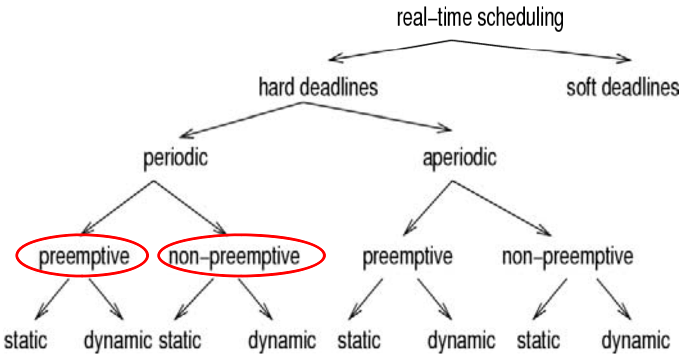
static

dynamic

static

dynamic

Preemptive/non-preemptive scheduling



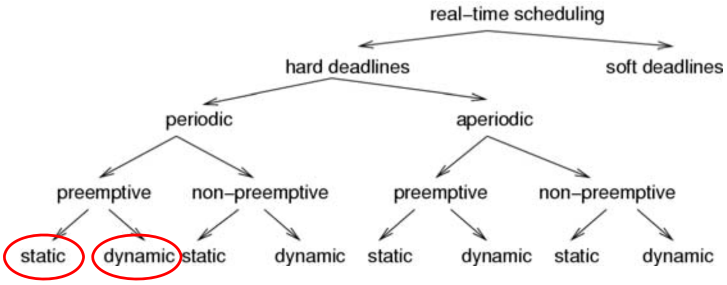
- **Non-preemptive schedulers:**

  Tasks are executed until they are done.

  Response time for external events may be quite long.
- **Preemptive schedulers:** To be used if
  - some tasks have long execution times or
  - if the response time for external events to be short.

Dynamic/online scheduling

- **Dynamic/online scheduling:**
  Processor allocation decisions (scheduling) at run-time; based on the information about the tasks arrived so far.

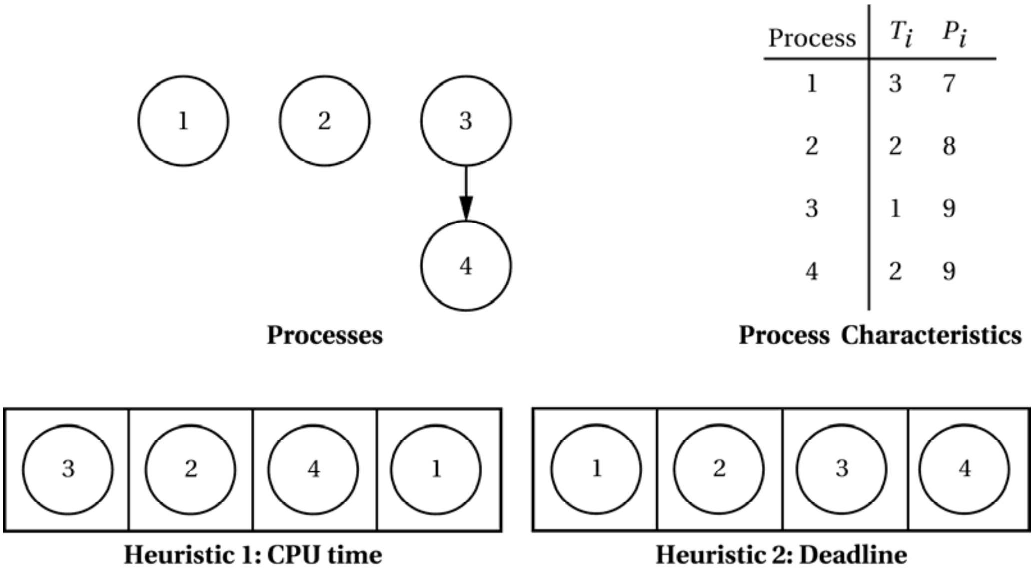


Static scheduling algorithms

- Often take advantage of data dependencies.
  - Resource dependencies come from the implementation.
- As-soon-as-possible (ASAP): schedule each process as soon as data dependencies allow.
- As-late-as-possible (ALAP): schedule each process as late as data dependencies and deadlines allow.

List scheduling

- A common form of constructive scheduler.



Processes

Process	T_i	P_i
1	3	7
2	2	8
3	1	9
4	2	9

Process Characteristics

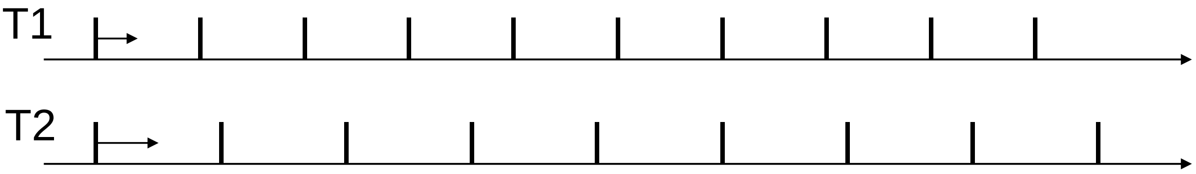
Heuristic 1: CPU time

Heuristic 2: Deadline

Priority-driven scheduling

- Each process has a priority.
- Processes may be ready or waiting.
- Highest-priority ready process runs in the current quantum.
    - Assume that lower-numbered processes have higher priority
- Priorities may be static or dynamic.

Periodic scheduling

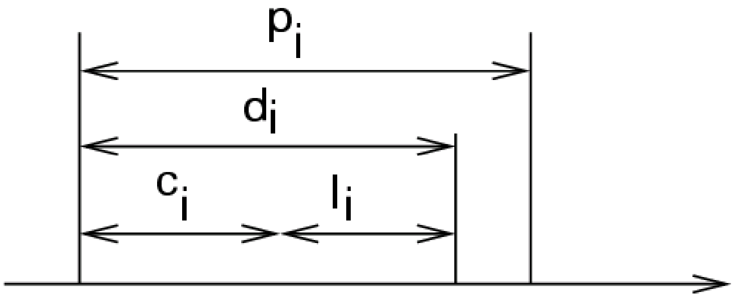


- For periodic scheduling, the best that we can do is to design an algorithm which will always find a schedule if one exists.

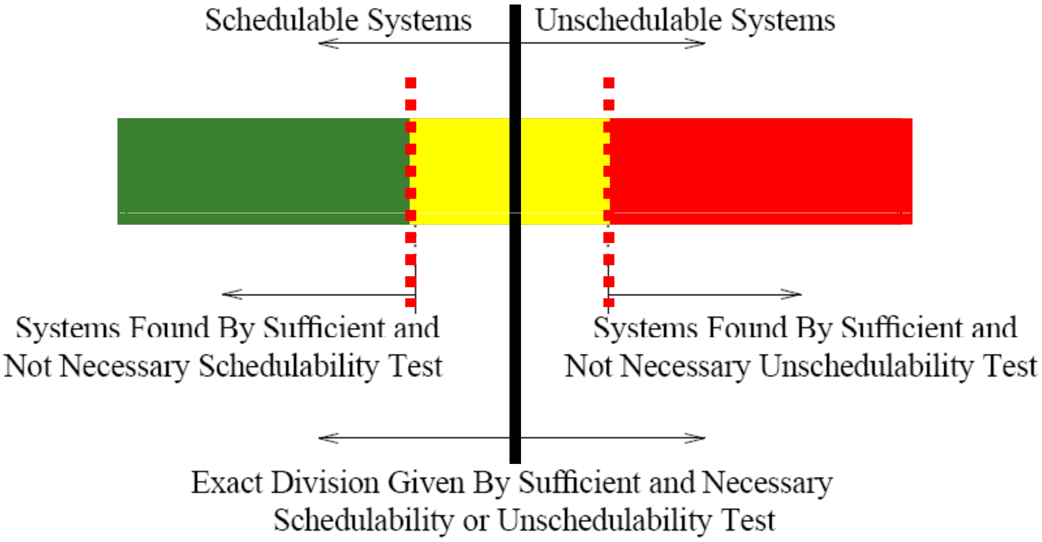
☞ A scheduler is defined to be **optimal** iff it will find a schedule if one exists.

Periodic scheduling

- Let
    - p_i be the period of task T_i,
    - c_i be the execution time of T_i,
    - d_i be the deadline interval, that is, the time between a job of T_i becoming available and the time after which the same job T_i has to finish execution.
    - l_i be the **laxity** or **slack**, defined as $l_i = d_i - c_i$



Schedulabilty test



Schedulable Systems

Unschedulable Systems

Systems Found By Sufficient and
Not Necessary Schedulability Test

Systems Found By Sufficient and
Not Necessary Unschedulability Test

Exact Division Given By Sufficient and Necessary
Schedulability or Unschedulability Test

Rate-monotonic scheduling (RMS)

- Liu and Layland: proved properties of static priority scheduling.
    - No data dependencies between processes.
    - Process periods may have arbitrary relationships.
    - Ideal (zero) context switching time.
    - Release time of process is start of period.
    - Process execution time is fixed.

Independent tasks:
Rate monotonic (RM) scheduling

- Most well-known technique for scheduling independent periodic tasks [Liu, 1973].
- **Assumptions:**
    - All tasks that have hard deadlines are periodic.
    - All tasks are independent.
    - $d_i = p_i$, for all tasks.
    - c_i is constant and is known for all tasks.
    - The time required for context switching is negligible.
    - For a single processor and for n tasks, the following equation holds for the accumulated utilization μ:

$$\mu = \sum_{i=1}^{n} \frac{c_i}{p_i} \le n(2^{1/n} - 1) \le \ln 2 \cong 0.6931$$

Rate monotonic (RM) scheduling
- The policy -

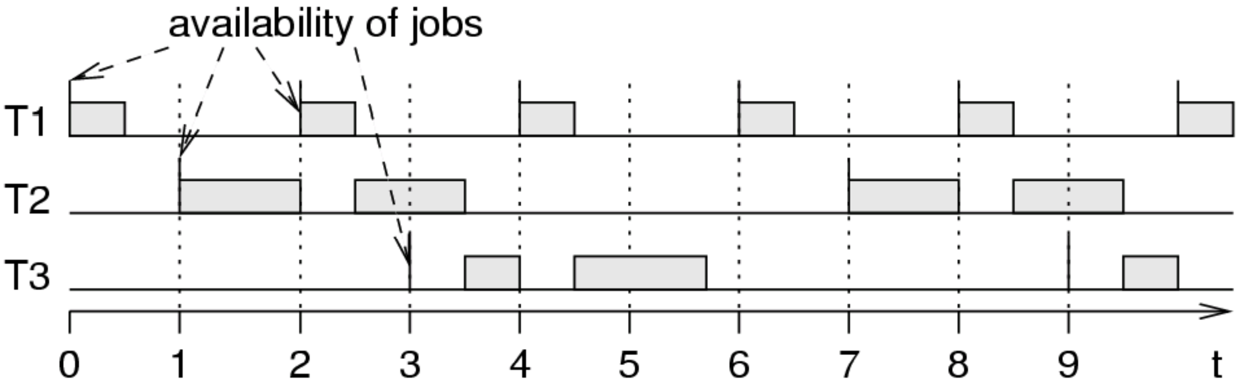
RM policy: The priority of a task is a monotonically decreasing function of its period. At any time, a highest priority task among all those that are ready for execution is allocated.

Theorem: If all RM assumptions are met, schedulability is guaranteed.

Schedulability test for RM

- **Approximate:** n tasks are guaranteed schedulable if $U \leq n(2^{1/n} - 1)$.
- **Precise:** n tasks are guaranteed schedulable iff, for all $1 \leq i \leq n$, we have $R_i \leq T_i$, where R_i is the worst-case response time.

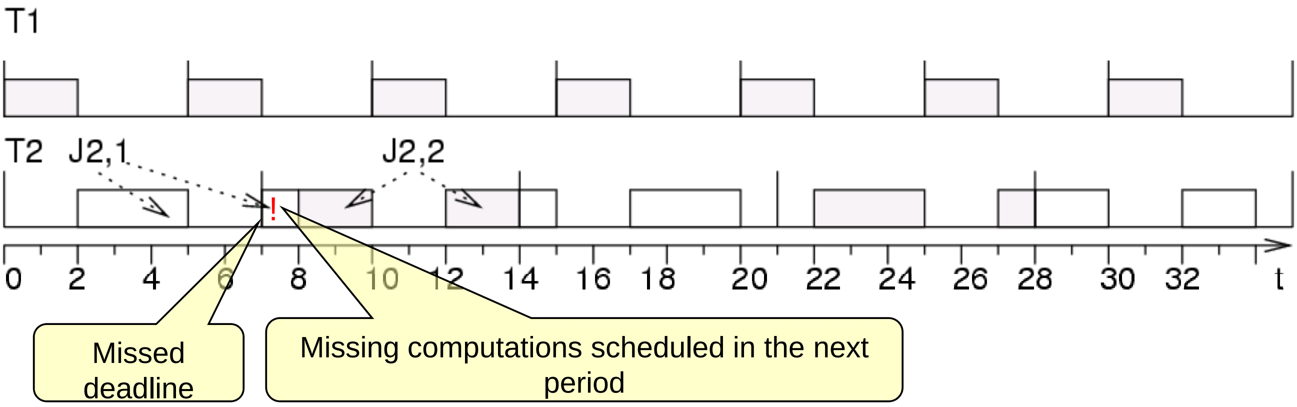
Example of RM-generated schedule



T1 preempts T2 and T3.
T2 and T3 do not preempt each other.

Case of failing RM scheduling

Task 1: period 5, execution time 2
Task 2: period 7, execution time 4
$\mu=2/5+4/7=34/35 \approx 0.97$
$2(2^{1/2}-1) \approx 0.828$



Missed deadline

Missing computations scheduled in the next period

Proof of RM optimality

- **Definition:** A **critical instant** of a task is the time at which the release of a task will produce the largest response time (worst-case response time).

- **Lemma:** For any task, the **critical instant** occurs if that task is simultaneously released with all higher priority tasks.

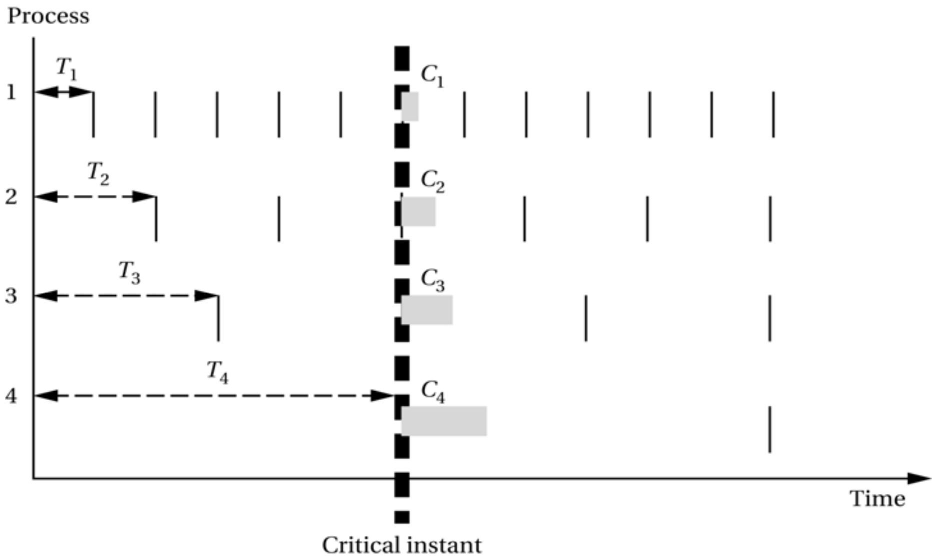
- **Proof:** Let $T=\{T_1, \ldots, T_n\}$: periodic tasks with $\forall i$: $p_i \leqq p_{i+1}$.

Critical instant

- The worst case combination of process executions that will cause the longest delay for the initiation of a process
- The critical instant of process i occurs when all higher priority processes are ready to execute
  - That is , when the deadlines of higher priority processes have just expired and new period have begun.

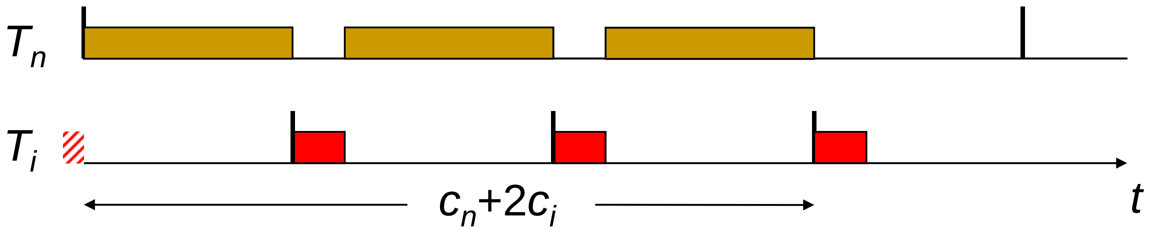
Critical instant

- Critical instant for process 4 occurs when processes 1,2, and 3 become ready.; the first three processes must run to completion before process 4 can start executing.

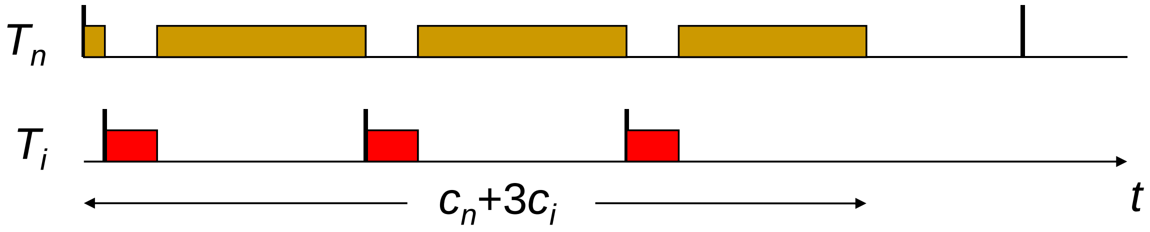


Critical instances (1)

- Response time of T_n is delayed by tasks T_i of higher priority:



- Delay may increase if T_i starts earlier



- Maximum delay achieved if T_n and T_i start simultaneously.

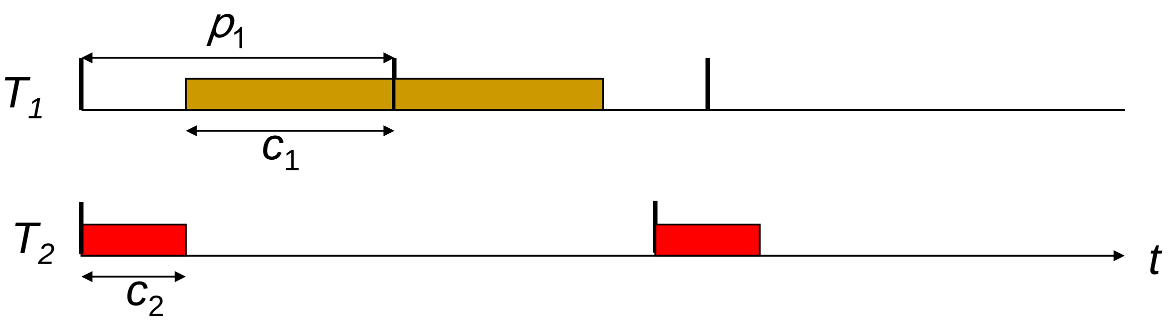
Critical instances (2)

- Repeating the argument for all $i = 1, \ldots n-1$:
- The worst case response time of a task occurs when it is released simultaneously with all higher-priority tasks. q.e.d.

- Schedulability is checked at the critical instants.
- If all tasks of a task set are schedulable at their critical instants, they are schedulable at all release times.

Proof of the RM theorem

- Let $T=\{T_1, T_2\}$ with $p_1 < p_2$.
- Assume RM is **not** used → priority(T_2) is highest:



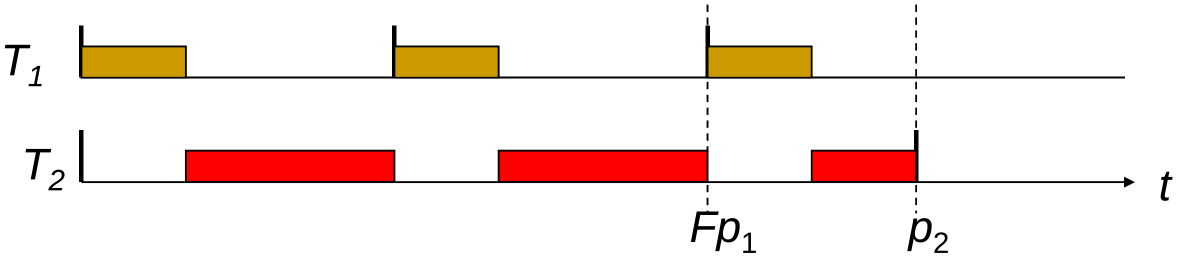
- Schedule is feasible if $c_1+c_2 \leqq p_1$ (1)

- Define $F=\lfloor p_2/p_1 \rfloor$: # of periods of T_1 fully contained in T_2

Proof of the RM theorem (2)

- Assume RM is used → priority(T_1) is highest:

- Case 1: $c_1 \le p_2 - Fp_1$
  (c_1 small enough to be finished before 2nd instance of T_2)



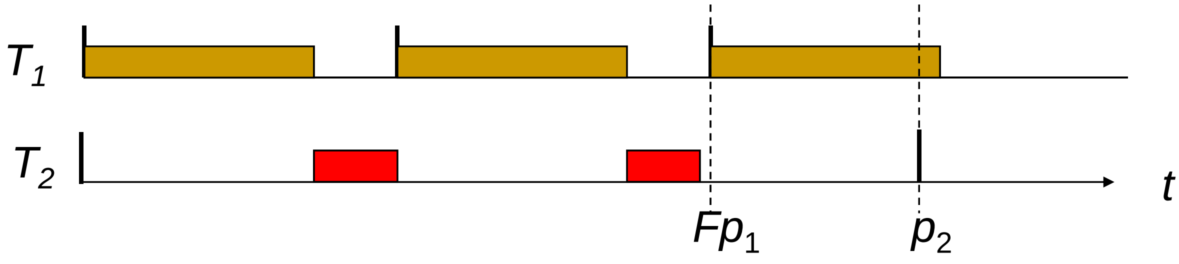
- Schedulable if $(F+1)\, c_1 + c_2 \le p_2$ (2)

Proof of the RM theorem (3)

- Not RM: schedule is feasible if $c_1+c_2 \le p_1$ (1)
- RM: schedulable if $(F+1)\, c_1 + c_2 \le p_2$ (2)
- From (1): $Fc_1+Fc_2 \le Fp_1$
- Since $F \ge 1$: $Fc_1+c_2 \le Fc_1+Fc_2 \le Fp_1$
- Adding c_1: $(F+1)c_1+c_2 \le Fp_1 + c_1$
- Since $c_1 \le p_2 - Fp_1$ (case 1): $(F+1)c_1+c_2 \le Fp_1 + c_1 \le p_2$
- Hence: if (1) holds, (2) holds as well
- ☞For case 1: Given tasks T_1 and T_2 with $p_1 < p_2$, then if the schedule is feasible by an arbitrary (but fixed) priority assignment, it is also feasible by RM.

Case 2: $c_1 > p_2 - Fp_1$

- Case 2: $c_1 > p_2 - Fp_1$
  (c_1 large enough not to finish before 2nd instance of T_2)



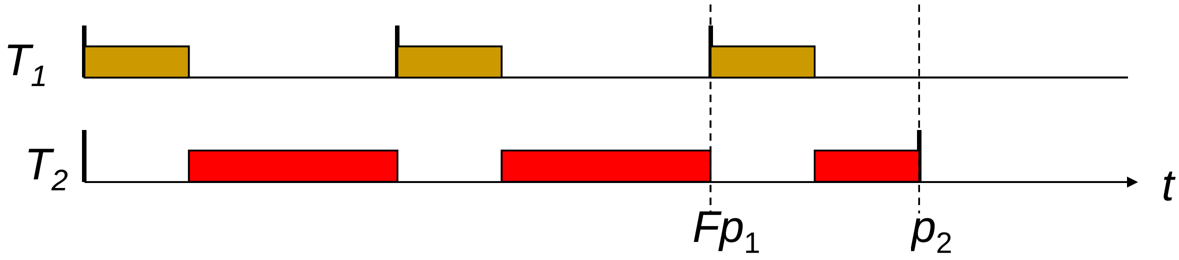
- Schedulable if $F c_1 + c_2 \le F p_1$ (3)
- $c_1 + c_2 \le p_1$ (1)
- Multiplying (1) by F yields $F c_1 + F c_2 \le F p_1$
- Since $F \ge 1$: $F c_1 + c_2 \le F c_1 + F c_2 \le F p_1$
- ☞ Same statement as for case 1.

Calculation of the least upper utilization bound

★

- Let $T=\{T_1, T_2\}$ with $p_1 < p_2$.
- Proof procedure: compute least upper bound U_{lup} as follows
  - Assign priorities according to RM
  - Compute upper bound U_{up} by setting computation times to fully utilize processor
  - Minimize upper bound with respect to other task parameters
- As before: $F=\lfloor p_2/p_1 \rfloor$
- c_2 adjusted to fully utilize processor.

Case 1: $c_1 \le p_2 - Fp_1$



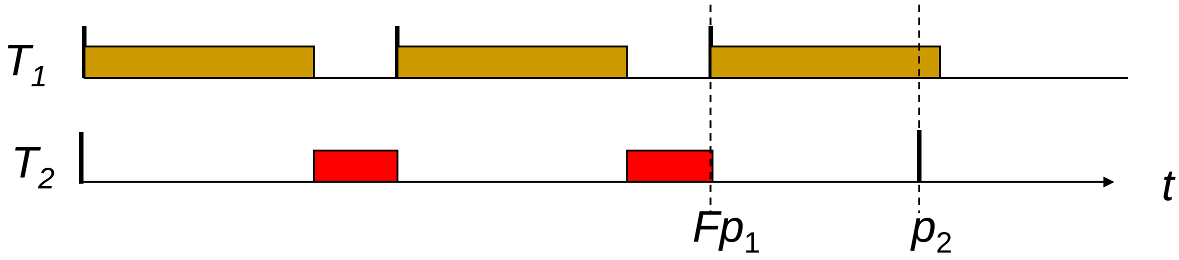
- Largest possible value of c_2 is $c_2 = p_2 - c_1 (F+1)$
- Corresponding upper bound is

$$U_{ub} = \frac{c_1}{p_1} + \frac{c_2}{p_2} = \frac{c_1}{p_1} + \frac{p_2 - c_1 (F+1)}{p_2} = 1 + \frac{c_1}{p_1} - \frac{c_1 (F+1)}{p_2} = 1 + \frac{c_1}{p_2}\left\{\frac{p_2}{p_1} - (F+1)\right\}$$

{ } is <0 → U_{ub} monotonically decreasing in c_1

Minimum occurs for $c_1 = p_2 - Fp_1$ (when *c*1 is maximum)

Case 2: $c_1 \ge p_2 - Fp_1$



- Largest possible value of c_2 is $c_2 = (p_1 - c_1)F$
- Corresponding upper bound is:

$$U_{ub} = \frac{c_1}{p_1} + \frac{c_2}{p_2} = \frac{c_1}{p_1} + \frac{(p_1 - c_1)\,F}{p_2} = \frac{p_1}{p_2}F + \frac{c_1}{p_1} - \frac{c_1}{p_2}F = \frac{p_1}{p_2}F + \frac{c_1}{p_2}\left\{\frac{p_2}{p_1} - F\right\}$$

{ } is ≥ 0 → U_{ub} monotonically increasing in c_1 (independent of c_1 if {}=0)

Minimum occurs for $c_1 = p_2 - Fp_{1,}$ (when *c*1 is minimum) as before.

Utilization as a function of $G=p_2/p_1-F$

- For $c1 = p2 - Fp1$:

$$U_{ub} = \frac{p_1}{p_2}F + \frac{c_1}{p_2}\left(\frac{p_2}{p_1} - F\right) = \frac{p_1}{p_2}F + \frac{\boxed{(p_2 - p_1F)}}{p_2}\left(\frac{p_2}{p_1} - F\right) = \frac{p_1}{p_2}\left\{F + \left(\frac{p_2}{p_1} - F\right)\left(\frac{p_2}{p_1} - F\right)\right\}$$

$$\text{Let } G = \frac{p_2}{p_1} - F; \qquad \Rightarrow \quad \boxed{\text{F: integer part, G: fractional part}}$$

$$U_{ub} = \frac{p_1}{p_2}\left(F + G^2\right) = \frac{\left(F + G^2\right)}{p_2/p_1} = \frac{\left(F + G^2\right)}{\left(p_2/p_1 - F\right) + F} = \frac{\left(F + G^2\right)}{F + G} = \frac{(F + G) - (G - G^2)}{F + G}$$

$$= 1 - \frac{G(1 - G)}{F + G}$$

Since $0 \leq G < 1$: $G(1-G) \geq 0$ → U_{ub} increasing in F →

Minimum of U_{ub} for min(F): F=1 →

$$U_{ub} = \frac{1 + G^2}{1 + G}$$

Proving the RM theorem for n=2

$$U_{ub} = \frac{1 + G^2}{1 + G}$$

Using derivative to find minimum of U_{ub} :

$$\frac{dU_{ub}}{dG} = \frac{2G(1 + G) - (1 + G^2)}{(1 + G)^2} = \frac{G^2 + 2G - 1}{(1 + G)^2} = 0$$

$$G_1 = -1 - \sqrt{2}; \qquad G_2 = -1 + \sqrt{2};$$

Considering only G_2, since $0 \leq G < 1$:

$$U_{lub} = \frac{1 + (\sqrt{2} - 1)^2}{1 + (\sqrt{2} - 1)} = \frac{4 - 2\sqrt{2}}{\sqrt{2}} = 2(\sqrt{2} - 1) = 2(2^{\frac{1}{2}} - 1) \cong 0.83$$

- This proves the RM theorem for the special case of n=2

Properties of RM scheduling

- From the proof, it is obvious that no idle capacity is needed if $p_2=F\,p_1$. In general: not required if the period of all tasks is a multiple of the period of the highest priority task, that is, schedulability is then also guaranteed if $\mu \leq 1$.
- RM scheduling is based on **static** priorities. This allows RM scheduling to be used in standard OS, such as Windows NT.
- A huge number of variations of RM scheduling exists.
- In the context of RM scheduling, many formal proofs exist.