correctness: $|C(t) - Cs(t)| < \epsilon$

drift is RoC of the clock value from perfect clock. Given clock has bounded drift ρ

$$| \frac{dC(t)}{dt} - 1 | < \rho$$

`fork()`

- create a `child` process that is identical to its parents, return `0` to child process and pid
- add a lot of overhead as duplicated. **Data space is not shared**

	OS	RTOS
philos	time-sharing	event-driven
requirements	high-throughput	schedulablity (meet all hard deadlines)
metrics	fast avg-response	ensureed worst-case response
overload	fairness	meet critical deadlines

Monotonicity: $\forall t_2 > t_1 : C(t_2) > C(t_1)$

preemption && `syscall`

> The act of temporarily interrupting a currently scheduled task for higher priority tasks.

> NOTE: `make` doesn't recompile if DAG is not changed.

process

- independent execution, logical unit of work scheduled by OS
- in virtual memory:
  - Stack: store local variables and function arguments
  - Heaps: dyn located (think of `malloc`, `calloc`)
  - BSS segment: uninit data
  - Data segment: init data (global & static variables)
  - text: RO region containing program instructions

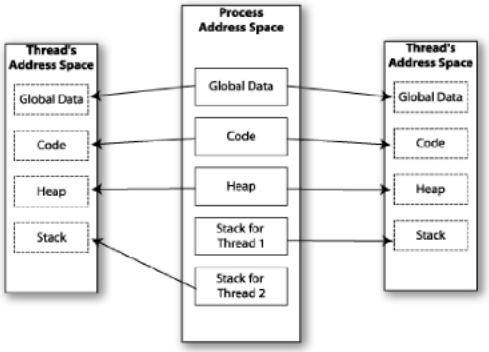
threads

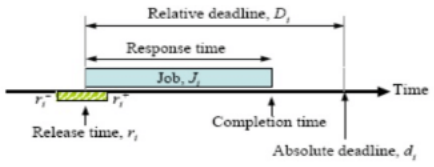
- program-wide resources: global data & instruction
- execution state of control stream
- shared address space for faster context switching

> - Needs synchronisation (global variables are shared between threads)
> - lack robustness (one thread can crash the whole program)

	interrupt	polling
speed	fast	slow
efficiency	good	poor
cpu-waste	low	high
multitasking	yes	yes
complexity	high	low
debug	difficult	easy

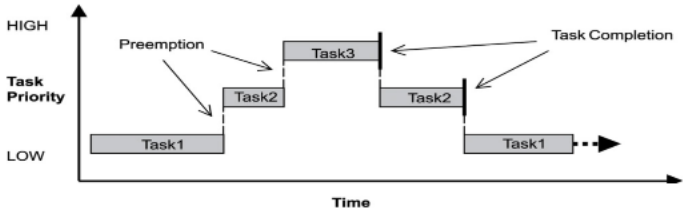
	stack	heap
creation	`Member m`	`Member*m = new Member()`
lifetime	function runs to completion	delete, free is called
grow	fixed	dyn added by OS
err	stack overflow	heap fragmentation
when	size of memory is known, data size is small	large scale dyn mem

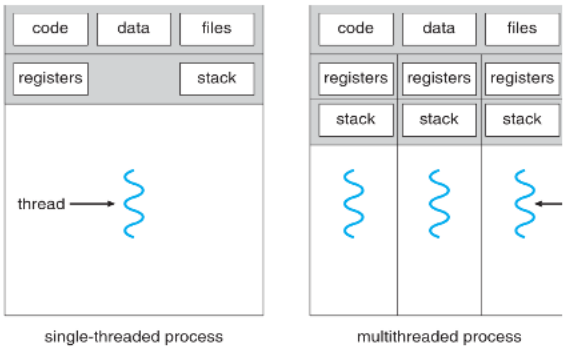




scheduling

1. Priority-based preemptive scheduling

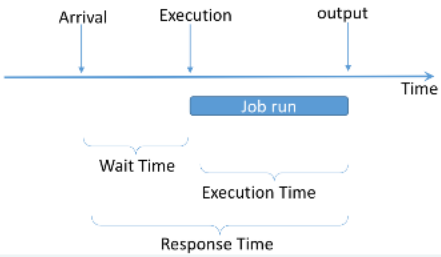




Temporal parameters:

Let the following be the scheduling parameters:

desc	var
# of tasks	n
release/arrival-time	$r_{i,j}$
absolute deadline	d_i
relative deadline	$D_i = r_{i,j} - d_i$
execution time	e_i
response time	R_i



Period p_i of a periodic task T_i is **min length** of all time intervales between release times of consecutive tasks.

Phase of a Task ϕ_i is the release time $r_{i,1}$ of a task T_i, or $\phi_i = r_{i,1}$

in phase are first instances of several tasks that are released simultaneously

Representation

a periodic task T_i can be represented by:

- 4-tuple (ϕ_i, P_i, e_i, D_i)
- 3-tuple (P_i, e_i, D_i), or $(0, P_i, e_i, D_i)$
- 2-tuple (P_i, e_i), or $(0, P_i, e_i, P_i)$

Utilisation factor u_i

for a task T_i with execution time e_i and period p_i is given by

$$u_i = \frac{e_i}{p_i}$$

For system with n tasks overall system utilisation is $U = \sum_{i=1}^{n} u_i$

cyclic executive

assume tasks are non-preemptive, jobs parameters with hard deadlines known.

- no race condition, no deadlock, just function call
- however, very brittle, number of frame F can be large, release times of tasks must be fixed

hyperperiod

> is the least common multiple (lcm)

maximum num of arriving jobs

$$N = \sum_{i=1}^{n} \frac{H}{p_i}$$

Frames: each task must fit within a single frame with size f => number of frames $F = \frac{H}{f}$

C1: A job must fit in a frame, or $f \geq \max e_i \forall\ 1 \leq i \leq n$ for all tasks

C2: hyperperiod has an integer number of frames, or $\frac{H}{f} = \text{integer}$

C3: $2f - \gcd(P_i, f) \leq D_i$ per task.

deadline-monotonic

- if every task has period equal to relative deadline, same as RM
- arbitrary deadlines then DM performs better than RM
- **RM always fails if DM fails**

Flow Graph for hyper-period

- Denote all jobs in hyperperiod of F frames as $J_1 \cdots J_F$
- Vertices:
  - k job vertices $J_1, J_2, \cdots, J_k$
  - F frame vertices $x, y, \cdots, z$
- Edges:
  - (source, J_i) with capacity $C_i = e_i$
    - Encode jobs' compute requirements
  - (J_i, x) with capacity f iff J_i can be scheduled in frame x
    - encode periods and deadlines
    - edge connected job node and frame node if the following are met:
      1. job arrives **before** or at the starting time of the frame
      2. job's absolute deadline **larger** or equal to ending time of frame
  - (f, sink) with capacity f
    - encodes limited computational capacity in each frame

rate-monotonic

- running on uniprocessor, tasks are preemptive, no OS overhead for preemption

task T_i has higher priority than task T_j if $p_i < p_j$

schedulability test for RM (Test 1)

Given n periodic processes, independent and preemptable, $D_i \geq p_i$ for all processes, **periods of all tasks are *integer* multiples of each other**

a sufficient condition for tasks to be scheduled on uniprocessor: $U = \sum_{i=1}^{n} \frac{e_i}{p_i} \leq 1$

schedulability test for RM (Test 2)

A *sufficient* but not necessary condition is $U \leq n \cdot (2^{\frac{1}{n}} - 1)$ for n periodic tasks

For $n \to \infty$, we have $U < \ln(2) \approx 0.693$

idea: find k such that time $t = k \times p_1 \geq k * e_1 + e_2$ and $k \times p_1 \leq p_2$ for task 2

general solution for RM-schedulability

The time demand function for task $i; 1 \leq i \leq n$:

$$\omega_i(t) \quad \sum_{k=1}^{i} \lceil \frac{t}{p_k} \rceil e_k \leq t$$

$\because 0 < t \leq p_i$

holds a time instant t chosen as $t = k_j p_j, (j = 1, \cdots, i)$ and $k_j = 1, \cdots, \lfloor \frac{p_i}{p_j} \rfloor$

earliest-deadline first (EDF)

depends on closeness of absolute deadlines

EDF schedulability test 1

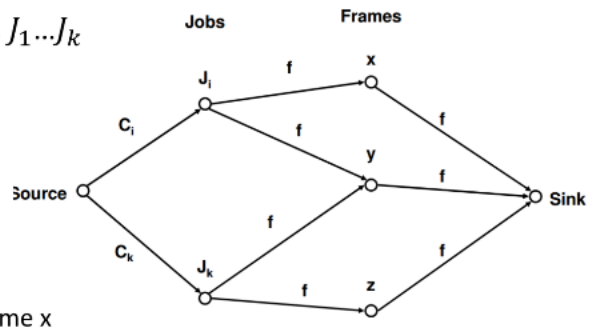
set of n periodic tasks, each whose relative deadline is equal to or greater than its period
iff $\sum_{i=1}^{n} (\frac{e_i}{p_i}) \leq 1$

EDF schedulability test 2

relative deadlines are not equal to or greater than their periods

$$\sum_{i=1}^{n} (\frac{e_i}{\min(D_i, p_i)}) \leq 1$$

s $J_1 \ldots J_k$



ame x

Priority Inheritance Protocol (PIP)

idea: increase the priorites only upon resource contention

avoid NPCS drawback

would still run into deadlock (think of RR task resource access)

Priority Inversion

critical sections to avoid **race condition**

> Higher priority task can be blocked by a lower priority task due to resource contention

shows how resource contention can delay completion of higher priority tasks

- access shared resources guarded by Mutex or semaphores
- access non-preemptive subsystems (storage, networks)

Resource Access Control

mutex

serially reusable: a resource cannot be interrupted

> If a job wants to use k_i units of resources R_i, it executes a lock $L(R_i; k_i)$, and unlocks $U(R_i; k_i)$ once it finished

Non-preemptive Critical Section Protocol (NPCS)

idea: schedule all critical sections non-preemptively

While a task holds a resource it executes at a priority higher than the priorities of all tasks

a higher priority task is blocked only when some lower priority job is in critical section

pros:
- zk about resource requirements of tasks

cons:
- task can be blocked by a lower priority task for a long time even without resource conflict

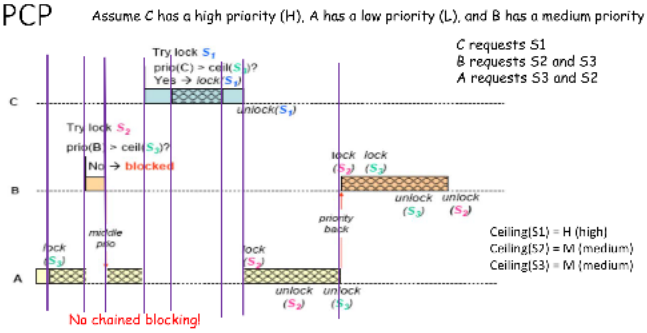
Priority Ceiling Protocol (PCP)

idea: extends PIP to prevent deadlocks
- assigned priorities are fixed
- resource requirements of all the tasks that will request a resource R is known

`ceiling(R)` : highest priority. Each resource has fixed priority ceiling

- Priority ceiling of a resource is fixed since we assume that we know all the resource requests. However, priority ceiling of the system is dynamically changed.
- When the priority of a task is updated?
  - When a task (T1) is blocked by another (T2) , we know that T2 is in its critical section which holds or will request a certain resource required by T1 (note that the contention occurs now or will occur in the future), so T2's priority is updated to the priority of T1.
- Only when a task has higher priority than the system's priority ceiling, the task will acquire a certain resource if it is available.

PCP



- A task can acquire a resource only if
  - the resource is free, AND
  - it has a higher priority than the priority ceiling of the system, or when the requesting task is holding the resource(s) whose priority ceiling is equal to the priority ceiling of the system.
- A task can be blocked by at most one critical section.
- Higher run-time overhead than Priority Inheritance Protocol