Real-Time Software (RTSOF)

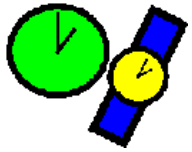
EVMINX9 Week 5

DE HAAGSE HOGESCHOOL

Scheduling Dienstregeling van taken

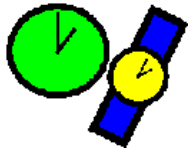
- N taken kun je in N! verschillende schedules uitvoeren.
  - Bijvoorbeeld 10 taken: 3628800 mogelijke schedules.
  - Met preemption nog veel meer mogelijkheden.
- De gekozen scheduling moet aan alle time requirements voldoen.
- Een scheduling scheme (=plan) bestaat uit:
  - Een algoritme om een schedule te bepalen.
  - Een methode om het “worst-case” gedrag van een, met het algoritme bepaalde, schedule te voorspellen.

Scheduling wanneer?



- Statisch: schedule ligt vast voor uitvoering.
  - Taken, worst-case executietijden en deadlines moeten van tevoren bekend zijn.
  - Je kunt door middel van een analyse aantonen dat alle deadlines worden gehaald.
  - Response tijden zijn voorspelbaar!
  - Kan niet reageren op “onvoorziene” situaties.
- Dynamisch: schedule bepaald tijdens uitvoeren.
  - Gedrag minder goed vooraf voorspelbaar.
  - Kan dynamisch inspelen op onvoorziene omstandigheden (b.v. berekening die langer duurt dan verwacht).

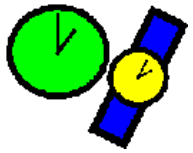
Scheduling RT systemen



- Bijna altijd wordt een statische methode gebruikt.

- Meest gebruikt: Preemptive Priority Based scheduling
  - op elk moment runt de ready taak met de hoogste prio
  - In dit geval bestaat een scheduling scheme uit:
    - Een algoritme om de prioriteit van elke taak te bepalen.
    - Een methode om de schedulability te voorspellen = Een methode om het “worst-case” gedrag bij de gekozen prioriteiten te voorspellen en te bekijken of aan alle time requirements wordt voldaan.

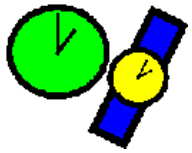
Scheduling Simpel beginnen



- Het aantal taken is bekend. N
- Alle taken zijn periodiek en de perioden zijn bekend. T_i
- De taken zijn onafhankelijk van elkaar (geen synchronisatie en communicatie).
- Systeem overhead wordt verwaarloosd.
- De deadline van elke taak is gelijk aan de periode. $D_i = T_i$
- De worst-case executietijd van elke taak is vast en bekend. C_i

Dit process model is **te** simpel (maar al moeilijk genoeg). Later zullen we realistische modellen bekijken.

Cyclic executive



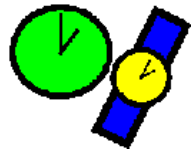
- Als schedule van tevoren bepaald is kun je het uitvoeren van de schedule expliciet programmeren.
- Voorbeeld:

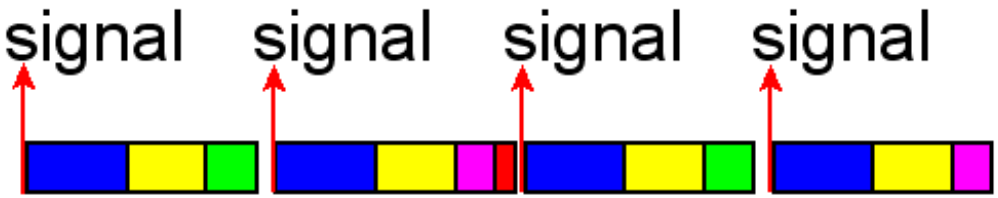
Taak	T	C
a	25	10
b	25	8
c	50	5
d	50	4
e	100	2

```
// zet timer: elke 25 ms een signal
while (1) {
    sigwait(&set, &signum); a(); b(); c();
    sigwait(&set, &signum); a(); b(); d(); e();
    sigwait(&set, &signum); a(); b(); c();
    sigwait(&set, &signum); a(); b(); d();
}
```

- Hoe bepaal je de schedulability?
- Hoe vind je een schedule?

Cyclic executive





a $T = 25$ $C = 10$

b $T = 25$ $C = 8$

c $T = 50$ $C = 5$

d $T = 50$ $C = 4$

e $T = 100$ $C = 2$

minor cycle = 25 ns

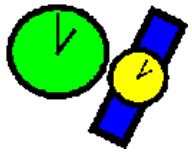
major cycle = 100 ns

Utilization (benutting) U = (4*10+4*8+2*5+2*4+1*2)/100 = **0,92**

Als C_e = 4 dan geldt U = **0,94** en er is **geen** schedule mogelijk!

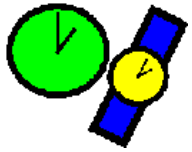
- Hoe bepaal je de schedulability?
- Hoe vind je een schedule?

Cyclic executive



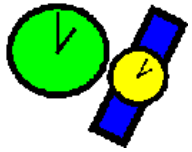
- Eigenschappen:
  - Er zijn geen processen (of threads) de taken zijn gewone functies.
  - Er is shared memory voor communicatie. Protectie is niet nodig.
  - Alle T 's moeten een veelvoud zijn van de minor cycle time.
  - Systeem is deterministisch.
- Problemen:
  - Taken met grote verschillen in T 's geeft lange major cycle.
  - Sporadic taken zijn niet in te passen!
  - Slecht onderhoudbaar, aanpasbaar en uitbreidbaar.
  - Schedule bepalen is moeilijk!
- Alternatieven:
  - **Fixed-Priority Scheduling** (FPS)
  - Earliest Deadline First (EDF)

FPS Fixed-Priority Scheduling



- Elke taak draait als thread (of process) met een statisch bepaalde vaste prioriteit.
- De prioriteit wordt bepaald door de time requirements.
- Preemptive ↔ NonPreemptive
  - Preemptive: Als thread met hogere prioriteit ready wordt dan wordt running thread meteen onderbroken.
  - NonPreemptive: Als thread eenmaal gestart is wordt hij ook afgemaakt.
  - Deferred Preemption: Als thread met hogere prioriteit ready wordt dan wordt running thread na een bepaalde tijd zeker onderbroken.

Rate Monotonic Priority Assignment

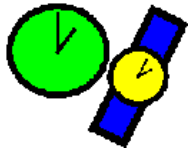


- De periode van een proces bepaalt de prioriteit van dat proces. Hoe korter de periode hoe hoger de prioriteit.

- $T_i < T_j \Rightarrow P_i > P_j$

- Deze methode is optimaal!
  - Dat wil zeggen dat als er een mogelijke preemptive FPS schedule is dan is rate monotonic FPS ook mogelijk.

FPS-RMPA



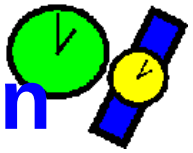
- Utilization based schedulability test:

$$U \equiv \sum_{i=1}^{N} \frac{C_i}{T_i} \leq N(2^{1/N} - 1)$$

N	Test
1	$U \leq 1.000$
2	$U \leq 0.828$
3	$U \leq 0.780$
4	$U \leq 0.757$
5	$U \leq 0.743$
10	$U \leq 0.718$
oneindig	$U \leq 0.693$

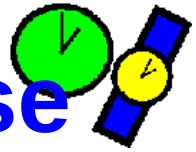
- Als deze test true geeft is worden geen deadlines gemist!
- Als deze test false worden er misschien deadlines gemist!

FPS-RMPA Schedulability voorbeelden



- Boek:
  - Voorbeeld A: Tabel 13.5, figuur 13.2 (time line) en figuur 13.5 (Grantt chart). Voldoet niet aan de test en haalt deadlines niet.
  - Voorbeeld B: Tabel 13.6. Voldoet wel aan de test en haalt deadlines wel.
  - Voorbeeld C: Tabel 13.7 en figuur 13.4. Voldoet niet aan de test maar haalt deadlines wel.
- Als aan de test voldaan wordt is dat voldoende bewijs dat de deadlines gehaald worden. Maar het is niet noodzakelijk dat aan de test voldaan wordt om de deadlines te kunnen halen.

FPS-RMPA Response tijd analyse



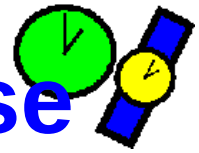
- In tegenstelling tot de vorige test geeft deze analyse de exacte response tijden. We kunnen dus exact zeggen of alle deadlines gehaald worden (en met welke marge).

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

R_i komt zowel **links** als **rechts** voor. De vergelijking is niet eenvoudig op te lossen (door de ceiling functie)

R_i is de response tijd van taak ***i***
hp(i) is de verzameling taken met een hogere prioriteit dan taak ***i***

FPS-RMPA Response tijd analyse

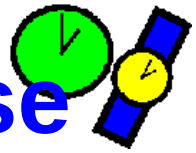


- Voor de taak met de hoogste prioriteit geldt: $R = C$
- Alle andere taken kunnen onderbroken worden en voor deze taken geldt: $R_i = C_i + I_i$
- Waarbij I_i de maximale "interference" is. Deze treed op als alle taken met een hogere prioriteit gelijk met taak ***i*** starten.
- Het aantal keer dat een taak met een hogere prioriteit ***j*** gestart wordt voordat ***i*** is afgelopen is:

$$\text{Number of Releases} = \left\lceil \frac{R_i}{T_j} \right\rceil$$

- $I_{i,j}$ is dus: $\left\lceil \frac{R_i}{T_j} \right\rceil C_j$

FPS-RMPA Response tijd analyse



- De totale maximale interference is de som van de maximale interference van alle taken met een hogere prioriteit:

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

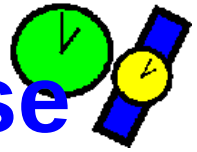
- Oplossing met behulp van recurrente betrekking:

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

Start met $w_i^0 = 0$ en ga door tot:

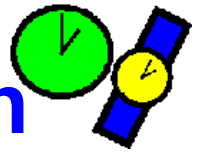
$w_i^n = w_i^{n+1}$ ✔ of $w_i^{n+1} > T_i$ ✘

FPS-RMPA Response tijd analyse



- Je kunt een programma schrijven om alle R_i's te berekenen: zie boek p. 476.
- Voorbeeld: zie boek p. 477.
- Verdere uitbreiding is nog nodig:
  - Taken met $D < T$.
  - Sporadic taken.
  - Interactie tussen taken.
  - Deferred preemption.
  - Release jitter.
  - Taken met $D > T$.
  - Fault tolerance.
  - Release offset.

FPS-RMPA $D < T$ en Sporadic taken



- $D < T$:
  - Gebruik DMPA in plaats van RMPA (geeft DMPO):

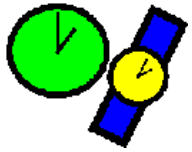
    $$D_i < D_j \Rightarrow P_i > P_j$$

  - Gebruik bij response tijd analyse als stop voorwaarde:

    $$W_i^{n+1} > D_i$$

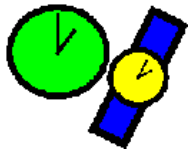
- Sporadic taken:
  - Vul voor de periodetijd de minimale tijd tussen twee "starts" van deze taak in. T = minimum inter-arrival interval.
  - Vaak geldt voor sporadic taken $D < T$.

FPS-DMPO Blocking



- Als een taak met een hoge prioriteit moet wachten op een taak met een lagere prioriteit dan wordt deze taak blocked (priority inversion).
- Een taak die unblocked sluit achter de taken met dezelfde prioriteit in de readyqueue aan.
- Als een taak met een lagere prioriteit moet wachten op een taak met een hogere prioriteit dan wordt deze taak preempted.
- Een taak die preempted wordt sluit voor de taken met dezelfde prioriteit in de readyqueue aan = vooraan de readyqueue!
- Om het real-time gedrag te kunnen voorspellen moet de maximale tijd dat een taak blocked kan zijn voorspelbaar zijn (bound blocking).

Priority inversion voorbeeld

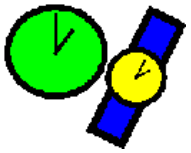


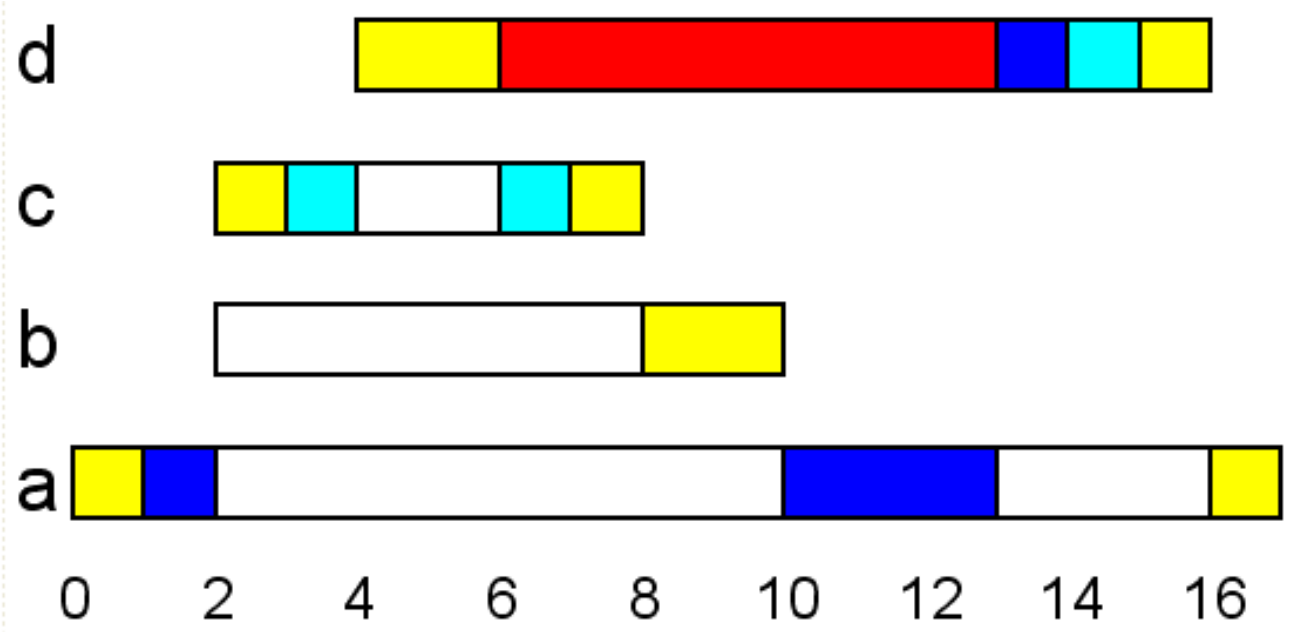
- 4 taken (a, b, c en d) delen 2 resources (Q en V).
- Deze resources kunnen slechts door 1 taak tegelijk gebruikt worden (en zijn dus beschermd met bijvoorbeeld een mutex).

taak	prio	execution	release time
d	4	EEQVE	4
c	3	EVVE	2
b	2	EE	2
a	1	EQQQQE	0

E = taak heeft alleen processor nodig
Q = taak heeft resource Q nodig
V = taak heeft resource V nodig

Priority inversion voorbeeld

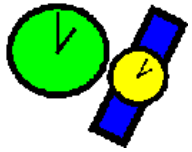




- Executing
- Executing with Q locked
- Executing with V locked
- Preempted
- Blocked

taak	prio	execution	release time
d	4	EEQVE	4
c	3	EVVE	2
b	2	EE	2
a	1	EQQQQE	0

FPS-DMPO Priority inversion

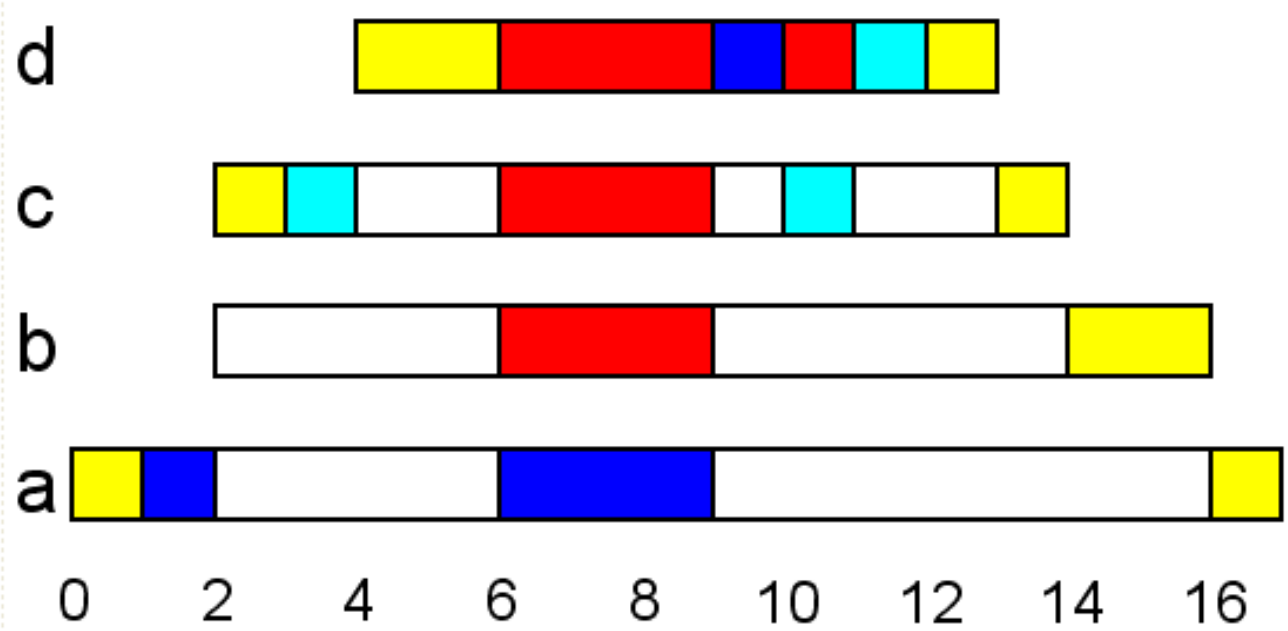


- Taak d wordt blocked door taak a, b en c (alle processen met een lagere prioriteit)!

- Blocking (priority inversion) is niet te voorkomen als we mutual exclusive recources delen.

- Blocking is wel te beperken door het toepassen van priority inheritance:
  - Als een taak blocked voor een resource dan krijgt de taak die de resource bezit de prioriteit van de taak die blocked wordt.

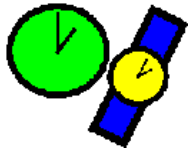
Priority inheritance voorbeeld



- Executing
- Executing with Q locked
- Executing with V locked
- Preempted
- Blocked

taak	prio	execution	release time
d	4	EEQVE	4
c	3	EVVE	2
b	2	EE	2
a	1	EQQQQE	0

Blocking Priority inheritance

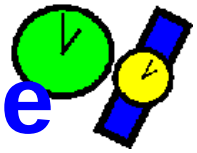


- De tijd dat een taak blocked kan zijn is nu beperkt.

$$B_i = \sum_{k=1}^{K} usage(k,i)C(k)$$

- ***B*** = maximum blocking time
- ***K*** = aantal gebruikte resources
- ***usage(k, i)*** = boolean functie
  - 1 als er een taak is met prioriteit lager dan P_i en een taak met prioriteit hoger of gelijk aan P_i (dit kan taak ***i*** zelf zijn) die de resource ***k*** delen.
- ***C(k)*** = maximale tijd dat resource ***k*** gebruikt wordt.

Blocking Response time analyse

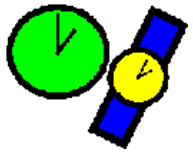


$$R_i = C_i + B_i + I_i$$

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

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

Blocking Priority inheritance

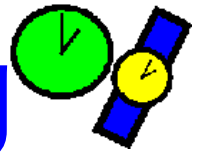


- Priority inheritance kan niet eenvoudig geïmplementeerd worden!
  - Bij semaphoren en condition variabelen is vaak niet te achterhalen op wie je wacht (wie de sem_post of pthread_cond_signal zal geven)!

  - Bij message passing is het vaak niet te achterhalen op wie je wacht (wie de mq_send zal doen waarop de mq_receive moet wachten)!

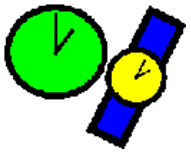
- Alternatief: priority ceiling

FPS-DMPO Priority ceiling

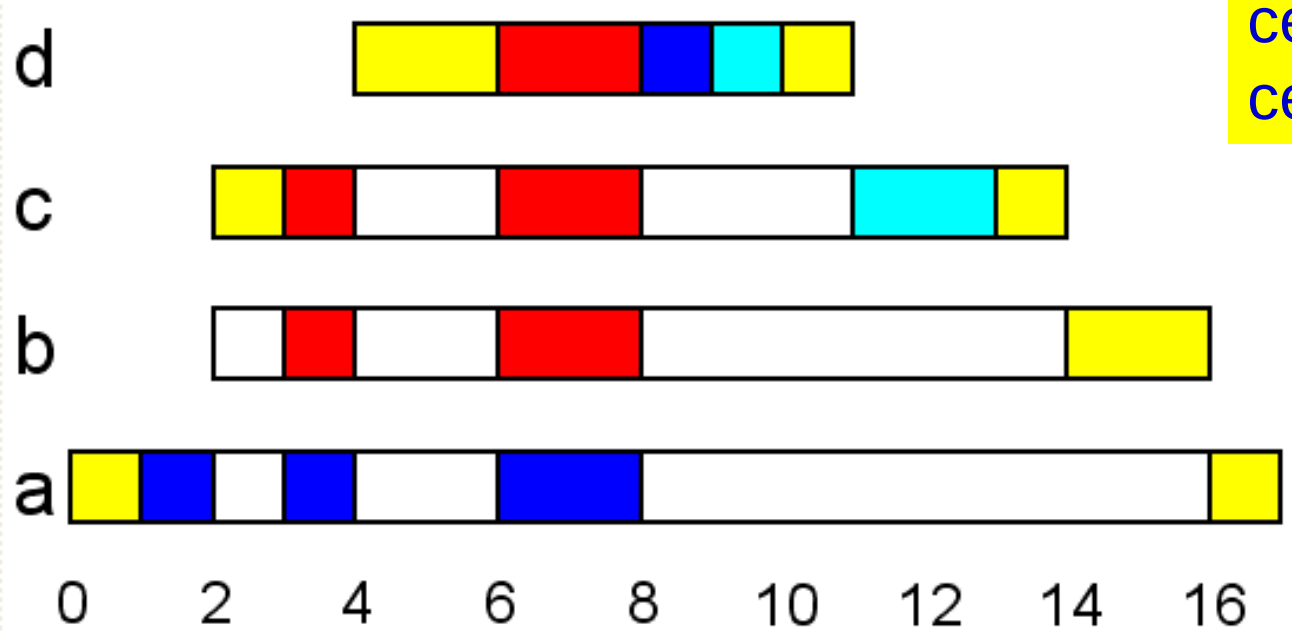


- Original Ceiling Priority Protocol OCPP:
  - Elke taak heeft een static priority.
  - Elke resource heeft een ceiling priority = maximale prioriteit van de processen die deze resource delen.
  - Een taak die een taak met een hogere prioriteit blocked erft de hogere prioriteit.
  - Een taak kan alleen een resource gebruiken als zijn prioriteit hoger is dan de ceiling van alle door andere taken gebruikte resources.
- Voordelen:
  - Proces kan maar 1x blocked worden.
  - Deadlock is niet mogelijk.
  - Mutual exclusive gaat altijd “van zelf” goed.
  - Transitive blocking is niet mogelijk.

OCPP voorbeeld



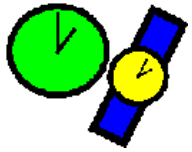
ceiling Q = 4
ceiling V = 4



Executing
Executing with Q locked
Executing with V locked
Preempted
Blocked

taak	prio	execution	release time
d	4	EEQVE	4
c	3	EVVE	2
b	2	EE	2
a	1	EQQQQE	0

OCPP Blocking

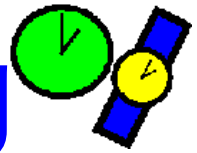


- Eerste resource kan altijd worden gebruikt.
- Tweede resource kan alleen worden gebruikt als er geen taak is met een hogere prioriteit die beide resources gebruikt.
- De tijd dat een taak blocked kan zijn wordt nu:

$$B_i = \max_{k=1}^{K} usage(k,i)C(k)$$

- Impementatie is nog steeds net zo moeilijk!
- Alternatief: Immediate CPP.

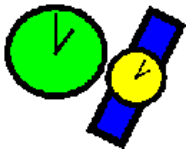
FPS-DMPO Priority ceiling

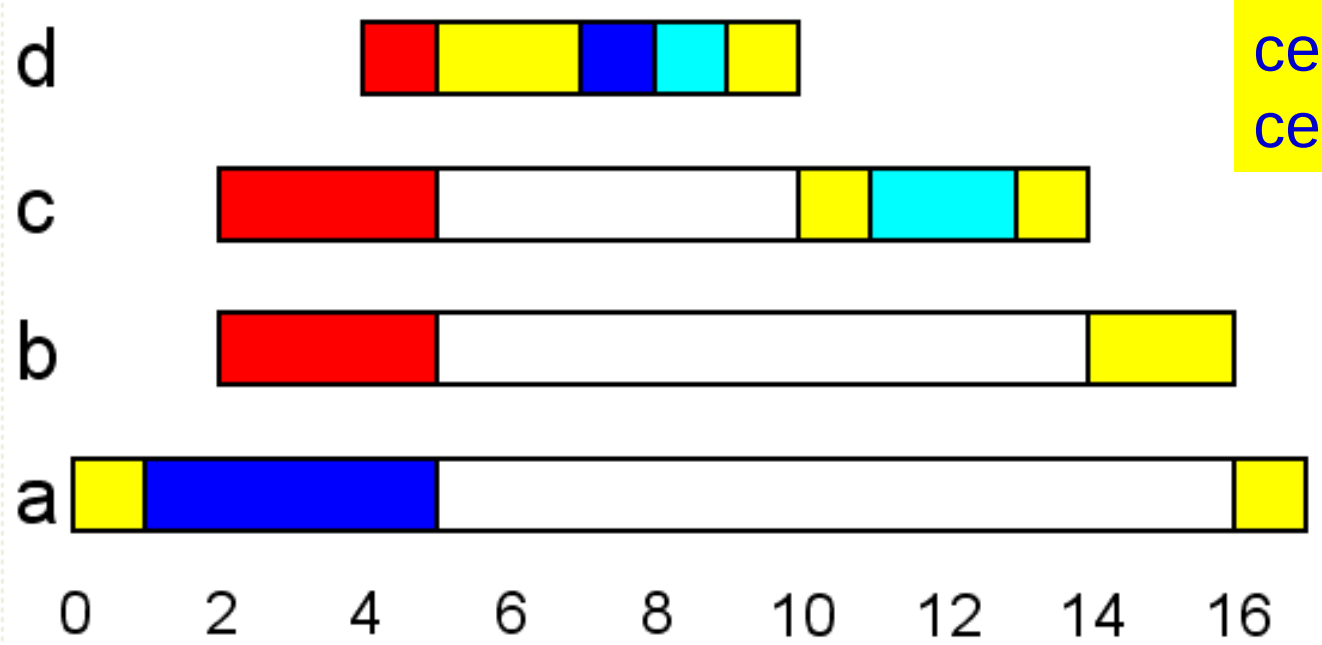


- Immediate Ceiling Priority Protocol ICPP:
  - Elke taak heeft een static priority.
  - Elke resource heeft een ceiling priority = maximale prioriteit van de processen die deze resource delen.
  - Als een taak een resource gebruikt krijgt die taak de ceiling prioriteit van de resource.

- Voordelen:
  - Proces kan maar 1x blocked worden.
  - Deadlock is niet mogelijk.
  - Mutual exclusive gaat altijd “van zelf” goed.
  - Transitive blocking is niet mogelijk.
  - Ten opzichte van OCPP:
    - Eenvoudiger te implementeren.
    - Minder taak wisselingen.

ICPP voorbeeld





ceiling Q = 4
ceiling V = 4

Executing
Executing with Q locked
Executing with V locked
Preempted
Blocked

taak	prio	execution	release time
d	4	EEQVE	4
c	3	EVVE	2
b	2	EE	2
a	1	EQQQQE	0

POSIX Priority Based Scheduling

- FIFO
  - Een thread blijft running totdat de thread blocked of preempted wordt.
  - Een thread die preempted wordt komt vooraan de readyqueue te staan.
- Round-Robbin
  - Een thread blijft running totdat de thread blocked of preempted wordt of totdat de time-slice is verstreken.
  - Een thread die preempted wordt komt vooraan de readyqueue te staan.
  - Een thread waarvan de time-slice is verstreken komt achter de threads met gelijke prioriteit op de readyqueue te staan.
- Sporadic Server
- Other

Priority Protect Protocol =ICPP

- int pthread_mutexattr_getprotocol(const thread_mutexattr_t* attr, int* protocol);
- int pthread_mutexattr_setprotocol(pthread_mutexattr_t *attr, int protocol);
  - mogelijke waarden voor protocol:
    - PTHREAD_PRIO_NONE
    - PTHREAD_PRIO_INHERIT
    - PTHREAD_PRIO_PROTECT

Bij een mutex weet je wel wie de pthread_mutex_unlock gaat doen!

POSIX naam voor ICPP

- int pthread_mutexattr_getprioceiling(const pthread_mutexattr_t* attr, int* prioceiling);
- int pthread_mutexattr_setprioceiling(pthread_mutexattr_t *attr, int prioceiling);