Negotiator Policy and Configuration

Greg Thain

Fairness in HTCondor and how to avoid it

Agenda

- Understand role of negotiator
- Learn how priorities work
- Learn how preemption works

- Encourage thought about possible policies!

After this talk, you should know..

Have a user get 2x cpus of another

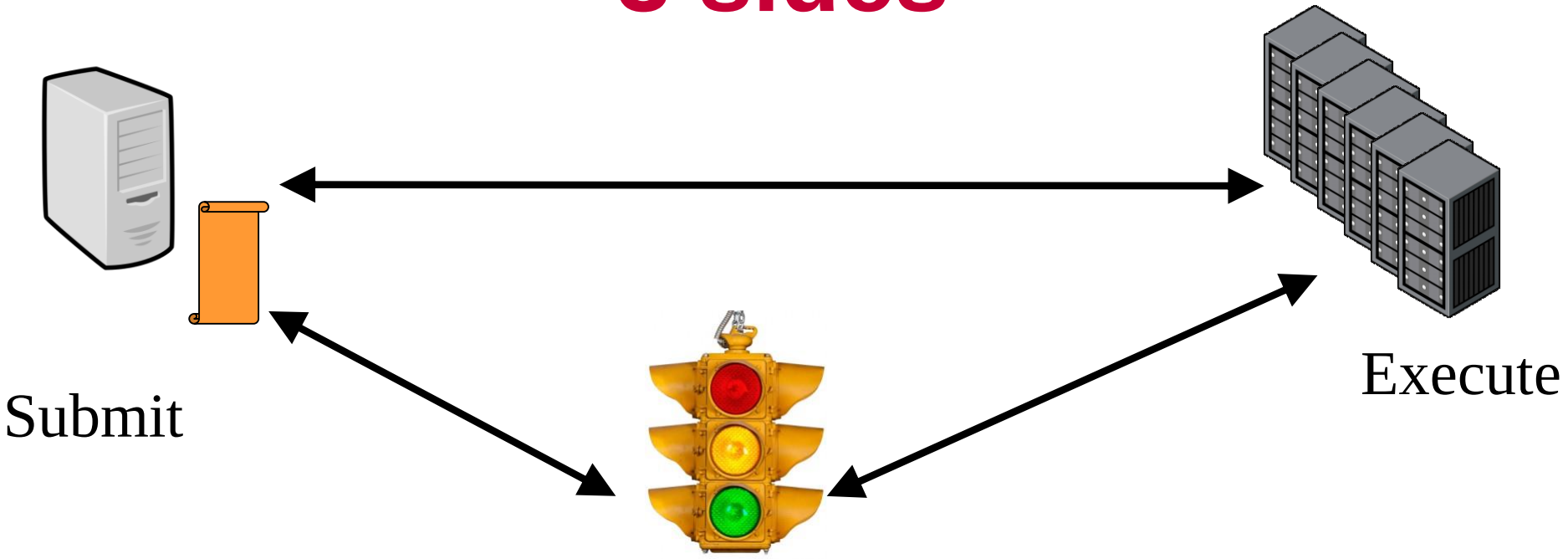
Schedule multicore jobs before single

Guarantee every job gets one hour runtime

Put a limit on licensed jobs in the pool

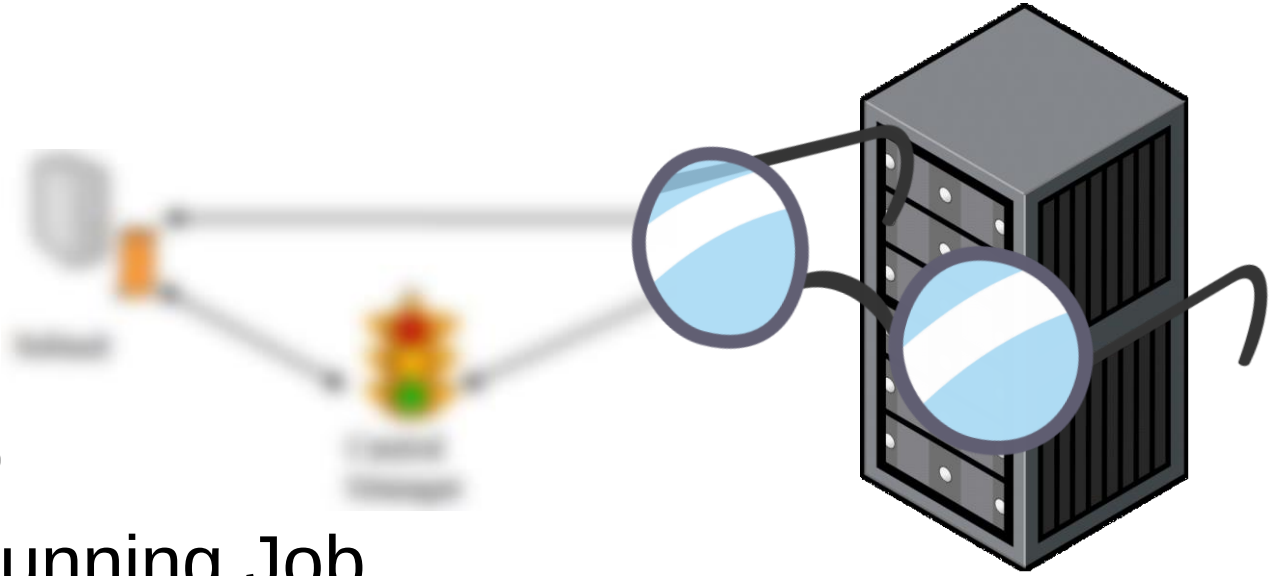
Three Truths and a Lie!

Overview of condor
3 sides



Startd Mission Statement

- Near sighted
- 3 inputs only:
  - Machine
  - Running Job
  - Candidate Running Job
- Knows nothing about the rest of the system!



Schedd mission

Run *jobs* on

slots the negotiator

has assigned to *submitters*.

Inputs:

All the jobs in that schedd

All the slots given to it by the negotiator

Schedd mission

Schedd Can:

- Re-use a slot for > 1 job (in succession)
- Pick which job for a submitter goes first

Schedd cannot:

- Reassign slots from one submitter to other

Submitter vs User

Submitters: what are they?

User: an OS construct

```
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
```

Submitter: Negotiator construct

- condor_userprio output
- ***submitters*** used in accounting and scheduling

1 Owner: 1 submitter

```
Executable = somejob
Universe = vanilla
…
queue
```

Submit UID	“Owner”	“Submitter”
gthain	gthain	gthain@UID_DOMAIN

1 Owner: 2 submitters

```
Executable = somejob
Universe = vanilla
nice_user = true
queue
```

Submit UID	“Owner”	“Submitter”
gthain	gthain	**nice-user.gthain@UID_DOMAIN**

Negotiation Mission

Assign the *slots* of the whole pool

to *submitters* based on some *policy* that's 'fair'

Negotiator Inputs

All the slots in the pool

All the submitters in the pool

All the submitters' priorities and quotas

One request per submitter at a time

How the Negotiator Works

Periodically tries to:

Rebalance ratio *slots assigned to submitters*

Via preemption, if enabled

Via assigning empty slots if not

Negotiator is always a little out of date

Concurrency Limits

Simplest Negotiator (+ schedd) policy

Useful for pool wide, across user limits

Useful Concurrency Limits:

> 100 running NFS jobs crash my server

License server only allows X concurrent uses

Only want 10 database jobs running at once

Concurrency Limits: How to Configure

add to negotiator config file (condor_reconfig needed):

```
NFS_LIMIT = 100
DB_LIMIT = 42
LICENSE_LIMIT = 5
```

Concurrency Limits: How to use

Add to job ad

```
Executable = somejob
Universe = vanilla
…
ConcurrencyLimits = NFS
queue
```

Concurrency Limits: How to use

OR

```
Executable = somejob
Universe = vanilla
…
ConcurrencyLimits = NFS:4
queue
```

Concurrency Limits: How to use

Add to job ad

```
Executable = somejob
Universe = vanilla
…
ConcurrencyLimits = NFS,DB
queue
```

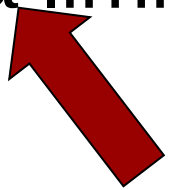
After this talk, you should know..

Have a user get 2x cpus of another

Schedule multicore jobs before single

Guarantee every job gets one hour runtime

Put a limit on licensed jobs in the pool



TRUTH!

Part of the picture

Concurrency limits very “strong”

Can throw off other balancing algorithms

No “fair share” of limits

Main Loop of Negotiation Cycle

1. Get all slots in the pool
2. Get all ~~jobs~~ submitters in pool
3. Compute # of slots submitters should get
4. In priority order, hand out slots to submitters
5. Repeat as needed

The Negotiator as Shell Script

1. Get all slots in the pool
2. Get all ~~jobs~~ submitters in pool
3. Compute # of slots submitters should get
4. In priority order, hand out slots to submitters
5. Repeat as needed

1: Get all slots in pool

1: Get all slots in pool

```
$ condor_status
```

1: Get ‘all’ slots in pool

```
NEGOTIATOR_SLOT_CONSTRAINT = some classad expr
```

NEGOTIATOR_SLOT_CONSTRAINT

Defaults to true:

Defines what subset of pool to use

For sharding, etc.

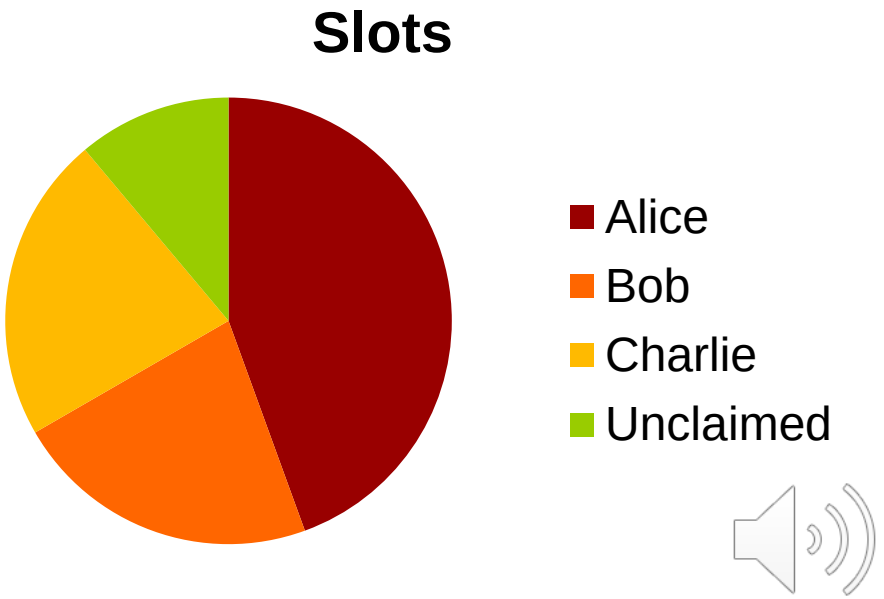
1: Get all slots in pool

```
$ condor_status -af Name State RemoteOwner
```

```
slot1@... Claimed Alice
slot2@... Claimed Alice
slot3@... Claimed Alice
slot4@... Unclaimed undefined
slot5@... Claimed Bob
slot6@... Claimed Bob
slot7@... Claimed Charlie
slot8@... Claimed Charlie
```

1: Get all slots in pool

```
$ condor_status -af Name RemoteOwner
```



2: Get all submitters in pool

```
$ condor_status -submitters
```

2: Get all submitters in pool

```
$ condor_status -submitters
```

```
Name     Machine RunningJobs IdleJobs
Alice    submit1 4                4
Bob      submit1 2              100
Charlie  submit1 2                0
Danny    submit1 0               50
```

2: Get all submitters in pool

```
$ condor_status -submitters
```

```
Name     Machine RunningJobs IdleJobs
Alice    submit1 4                4
Bob      submit1 2              100
Charlie  submit1 2                0
Danny    submit1 0               50
```

3:Compute per-submitter “share”

Tricky

Based on historical usage

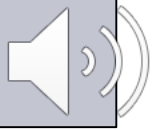
3a: Get historical usage

```
$ condor_userprio -all
```

3a: Get historical usage

```
$ condor_userprio -all
```

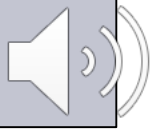
```
UserName Effective Real      Priority  Res
         Priority Priority    Factor  in use
Alice       3100      3.1      1000      4
Bob         4200      4.2      1000      2
Charlie     1500      1.5      1000      2
Danny       8200      8.2      1000      0
```



3a: Get historical usage

$$EffectivePrio = RealPrio \text{ X } PrioFactor$$

UserName	Effective Priority	Real Priority	Priority Factor	Res in use
Alice	3100	3.1	1000	4
Bob	4200	4.2	1000	2
Charlie	1500	1.5	1000	2
Danny	8200	8.2	1000	0



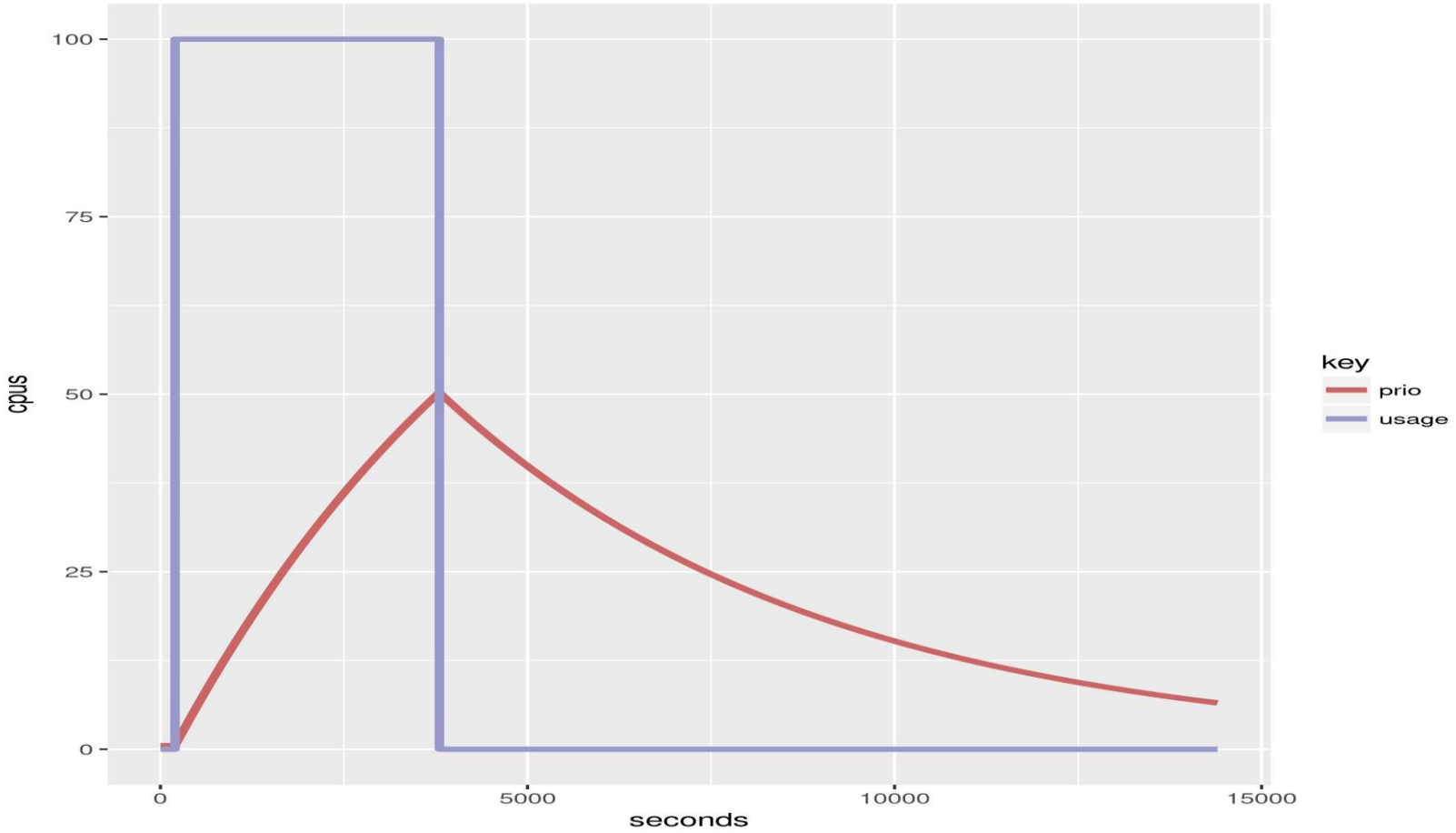
So What is Real Priority?

Real Priority is smoothed historical usage

Smoothed by `PRIORITY_HALFLIFE`

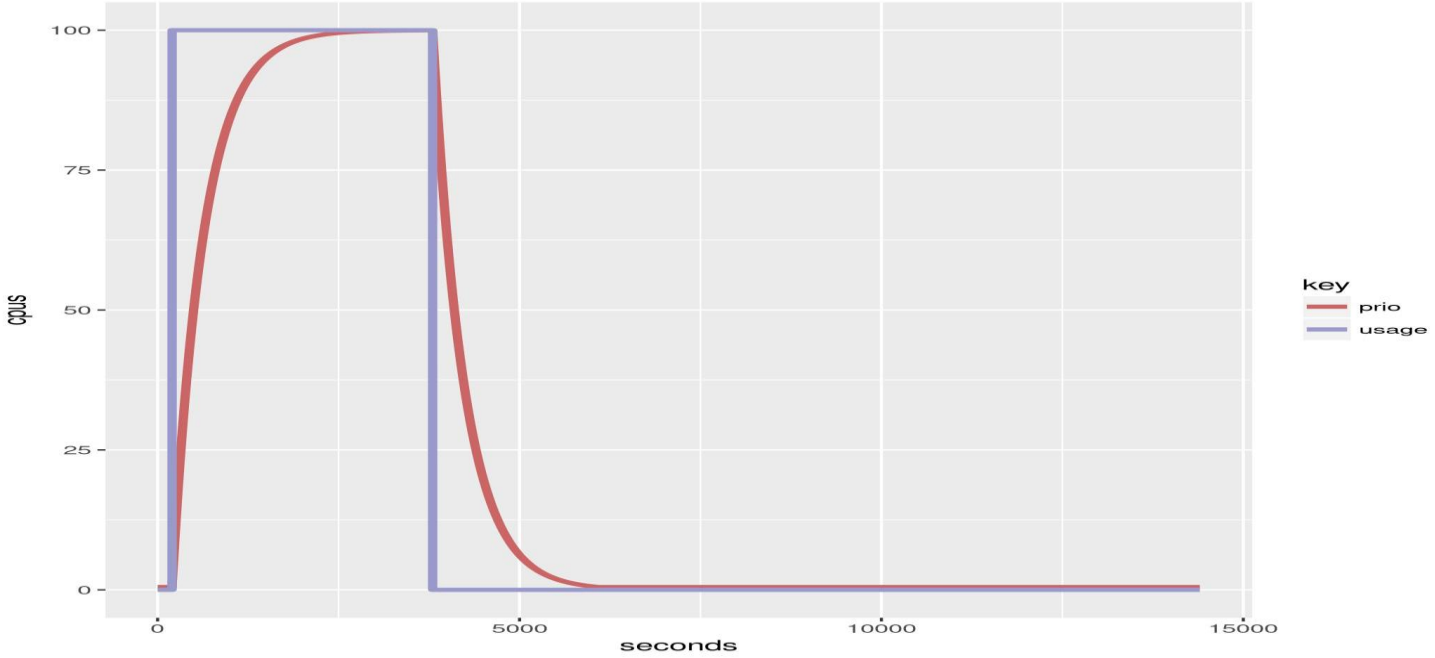
`PRIORITY_HALFLIFE` defaults 86400s (24h)

Actual Use vs Real Priority



Another PRIORITY_HALFLIFE

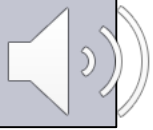
PRIORITY_HALFLIFE = 1



3a: Get historical usage

```
$ condor_userprio -all
```

```
UserName Effective Real      Priority  Res
         Priority Priority    Factor  in use
Alice       3100      3.1      1000      4
Bob         4200      4.2      1000      2
Charlie     1500      1.5      1000      2
Danny       8200      8.2      1000      0
```



Effective priority:

- Effective Priority is the *ratio* of the pool that the negotiator tries to allot to *submitters*

Lower is better, 0.5 is the best real priority

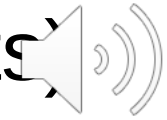
UserName	Effective Priority	Real Priority	Priority Factor	Res in use
Alice	1000	1.0	1000	4
Bob	2000	2.0	1000	2
Charlie	2000	2.0	1000	2

Alice deserves 2x Bob & Charlie

Alice: 4

Bob: 2

Charlie: 2 (Assuming 8 total slots)

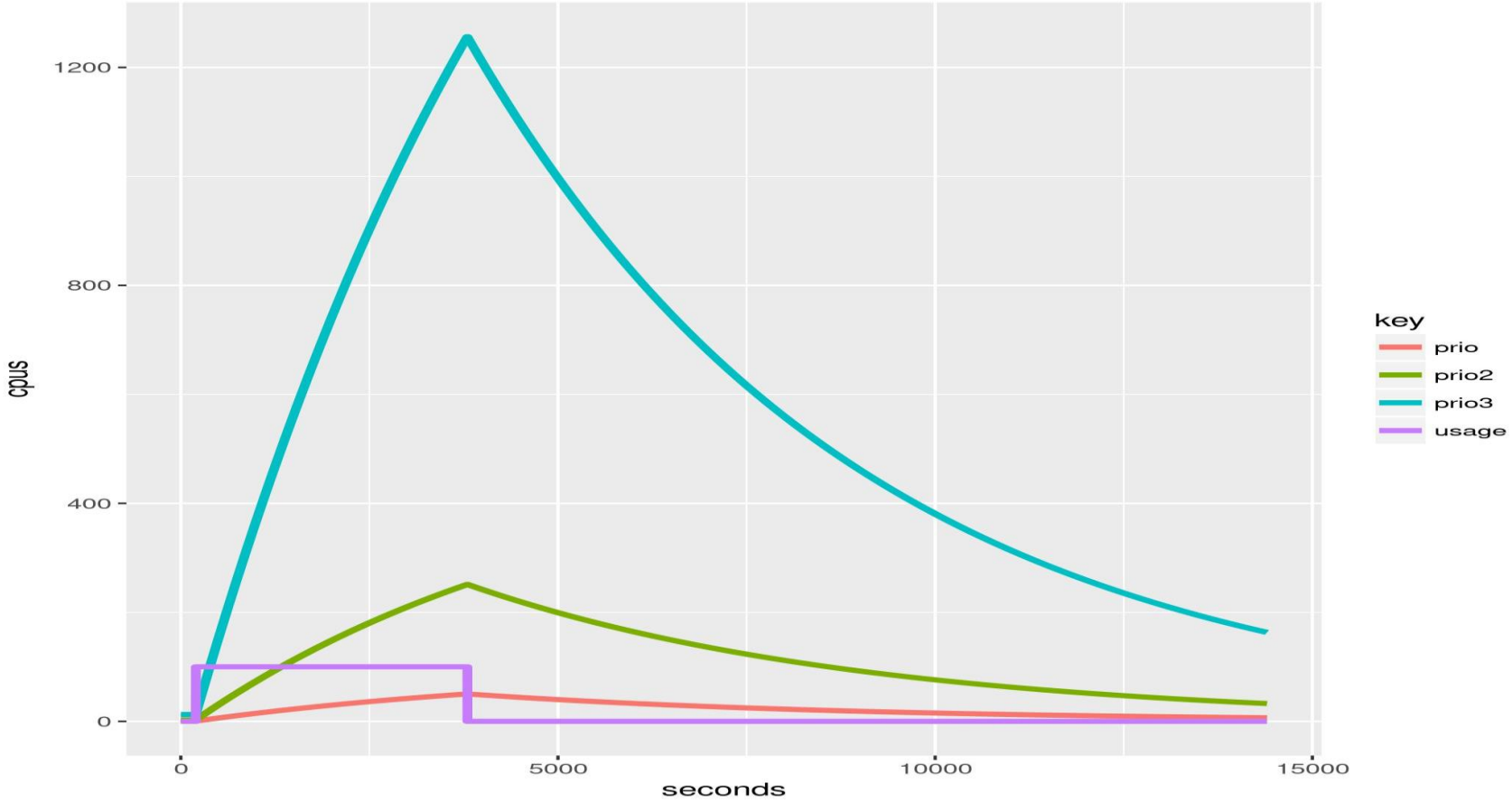


```
UserName Effective Real      Priority  Res
         Priority Priority   Factor  in use
Alice      1000      1.0      1000      4
Bob        2000      2.0      1000      2
Charlie    2000      2.0      1000      2
```

Priority factor lets admin say
If equal usage, User A gets 1/nth User B

```
$ condor_userprio –setfactor alice 5000
```

3 different PrioFactors



Priority Factor pop quiz

```
$ condor_userprio -setfactor alice 500
$ condor_userprio -setfactor bob   1000
```

Gives Alice 2x Bob

When both have jobs

Either Alice or Bob can use whole pool when other is gone

Whew! Back to negotiation

1. Get all slots in the pool
2. Get all ~~jobs~~ submitters in pool
3. Compute # of slots submitters should get
4. In priority order, hand out slots to submitters
5. Repeat as needed

Target allocation from before

User	Effective Priority	Goal
Alice	1,000.00	4
Bob	2,000.00	2
Charlie	2,000.00	2

Assume 8 total slots (claimed or not)

Look at current usage

User	Effective Priority	Goal	Current Usage
Alice	1,000.00	4	3
Bob	2,000.00	2	1
Charlie	2,000.00	2	0

Diff the goal and reality

User	Effective Priority	Goal	Current Usage	Difference (“Limit”)
Alice	1,000.00	4	3	1
Bob	2,000.00	2	1	1
Charlie	2,000.00	2	0	2

Limits determined, matchmaking starts

In Effective User Priority order,

Find a schedd for that user, get the request

User	Effective Priority	Difference (“Limit”)
Alice	1,000.00	**1**
Bob	2,000.00	**1**
Charlie	2,000.00	**2**

Three Truths and one Lie!

Have a user get 2x slots of another

~~Schedule multicore jobs before single~~

Guarantee every job gets one hour runtime

Put a limit on licensed jobs in the pool

“Requests”, not “jobs”

```
$ condor_q -autocluster Alice
Id      Count Cpus Memory Requirements
20701      10    1   2000  OpSys == “Linux”
20702      20    2   1000  OpSys == “Windows”
```

Match *all* machines to requests

```
Id      Count Cpus Memory Requirements
20701      10    1   2000  OpSys == "Linux"
```

```
slot1@... Linux   X86_64 Idle    2048
slot2@... Linux   X86_64 Idle    2048
slot1@... Linux   X86_64 Idle    1024
slot2@... Linux   X86_64 Claimed 2048
slot1@... WINDOWS X86_64 Claimed 1024
```

Sort All matches

By 3 keys, in order

```
NEGOTIATOR_PRE_JOB_RANK
RANK
NEGOTIATOR_POST_JOB_RANK
```

Why Three?

`NEGOTIATOR_PRE_JOB_RANK`

Strongest, goes first over job RANK

`RANK`

Allows User some say

`NEGOTIATOR_POST_JOB_RANK`

Fallback default

PRE_JOB_RANK use case

Policy:

“I want all my fast machines filled first”

```
NEGOTIATOR_PRE_JOB_RANK = mips
```

Finally, give matches away!

```
slot1@... Linux   X86_64 Unclaimed 2048
slot2@... Linux   X86_64 Unclaimed 2048
slot1@... Linux   X86_64 Claimed   2048
```

Up to the limit specified earlier

If below limit, ask for next job request

Done with Alice, on to Bob

User	Effective Priority	Difference (“Limit”)
Alice	1,000.00	**1**
Bob	2,000.00	**1**
Charlie	2,000.00	**2**

But, it isn’t that simple…

Assumed every job matches every slot

- And infinite supply of jobs!

… But what if they don’t match?

- There will be leftovers – then what?

Lather, rinse, repeat

This whole cycle repeats with leftover slots

Again in same order…

Big policy question

Preemption: Yes or no?

Tradeoff: fairness vs. throughput

(default: no preemption)

Preemption: disabled by default

```
PREEMPTION_REQUIREMENTS = false
```

Evaluated with slot & request ad. If true,

Claimed slot is considered matched, and

Subject to matching

Example PREEMPTION_REQs

```
PREEMPTION_REQUIREMENTS=\
 RemoteUserPrio > SubmitterPrio * 1.2
```

PREEMPTION_RANK

- Sorts matched preempting claims

```
PREEMPTION_RANK = -TotalJobRunTime
```

MaxJobRetirementTime

Can be used to guarantee minimum time

E.g. if claimed, give an hour runtime, no matter what:

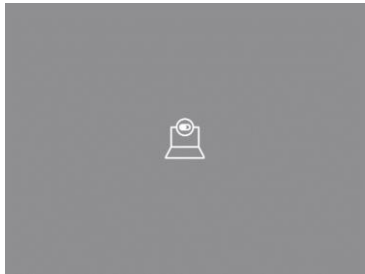
MaxJobRetirementTime = 3600

Can also be an expression

Submitter Ceiling

Upper bound on cpus any one user gets

$ condor_userprio –setceiling username 100



Three Truths and one Lie!

Have a user get 2x slots of another

~~Schedule multicore jobs before single~~

Guarantee every job gets one hour runtime

Put a limit on licensed jobs in the pool

Where to go for more help

htcondor.readthedocs.io
htcondor-users email list
htcondorproject.org

Thank You!