Open Science Grid

Intermediate HTCondor: Workflows

Monday pm

Greg Thain

Center For High Throughput Computing

University of Wisconsin-Madison

Before we begin…

Any questions up to now?

Quick Review: 1

1 Job

```
Executable = runme.sh

Output   = out
Error    = err
Log      = log
Request_Memory = 1024
…
queue
```

Quick Review: 2

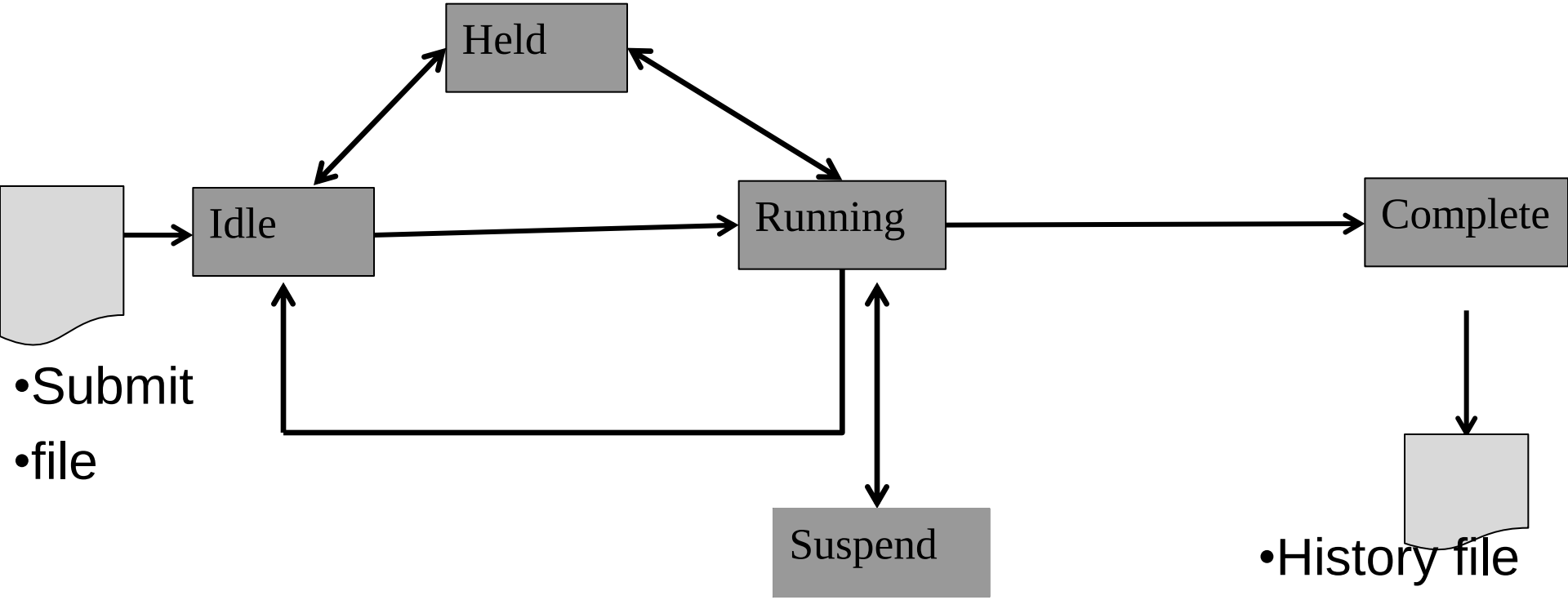
Many Jobs

```
Executable = runme.sh

Output  = out.$(PROCESS)
Error   = err.$(PROCESS)
Log     = log.$(PROCESS)
Request_Memory = 1024

Queue 10000
```

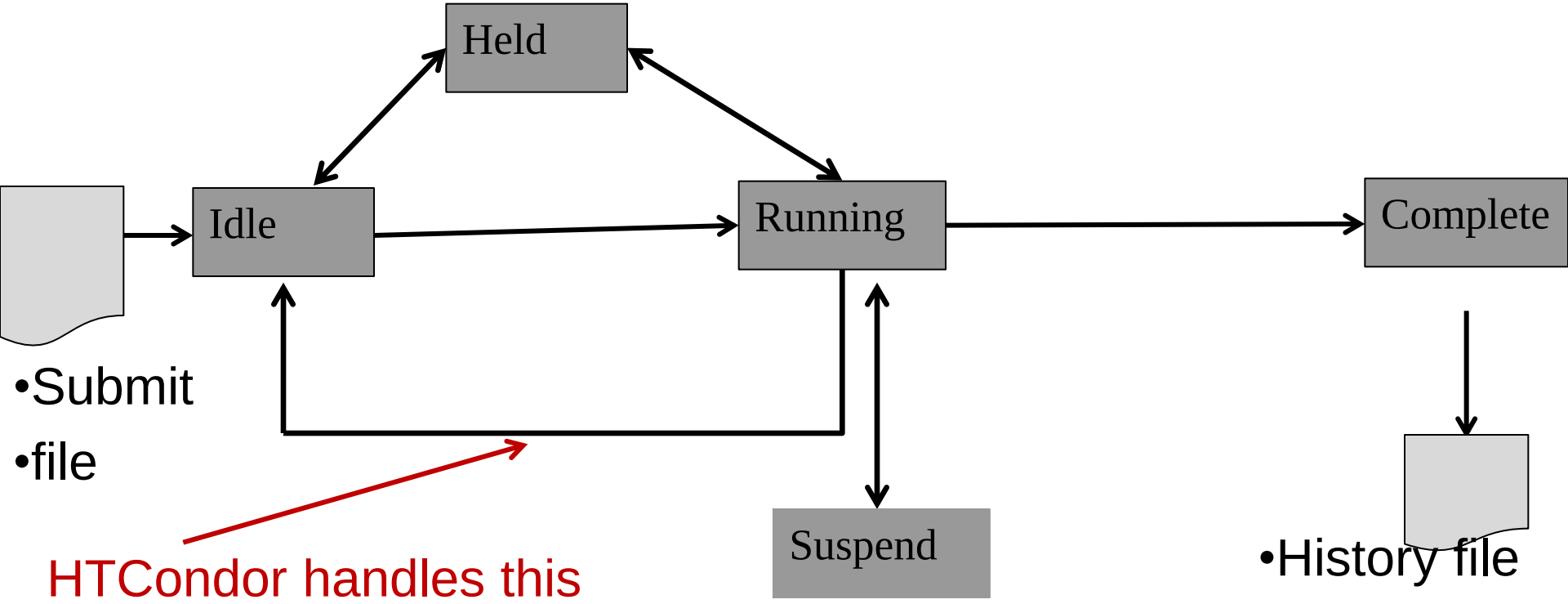
What could go wrong at scale?

- Machine has bad disk
- Machine has wrong OS version
- Machine has missing dependencies

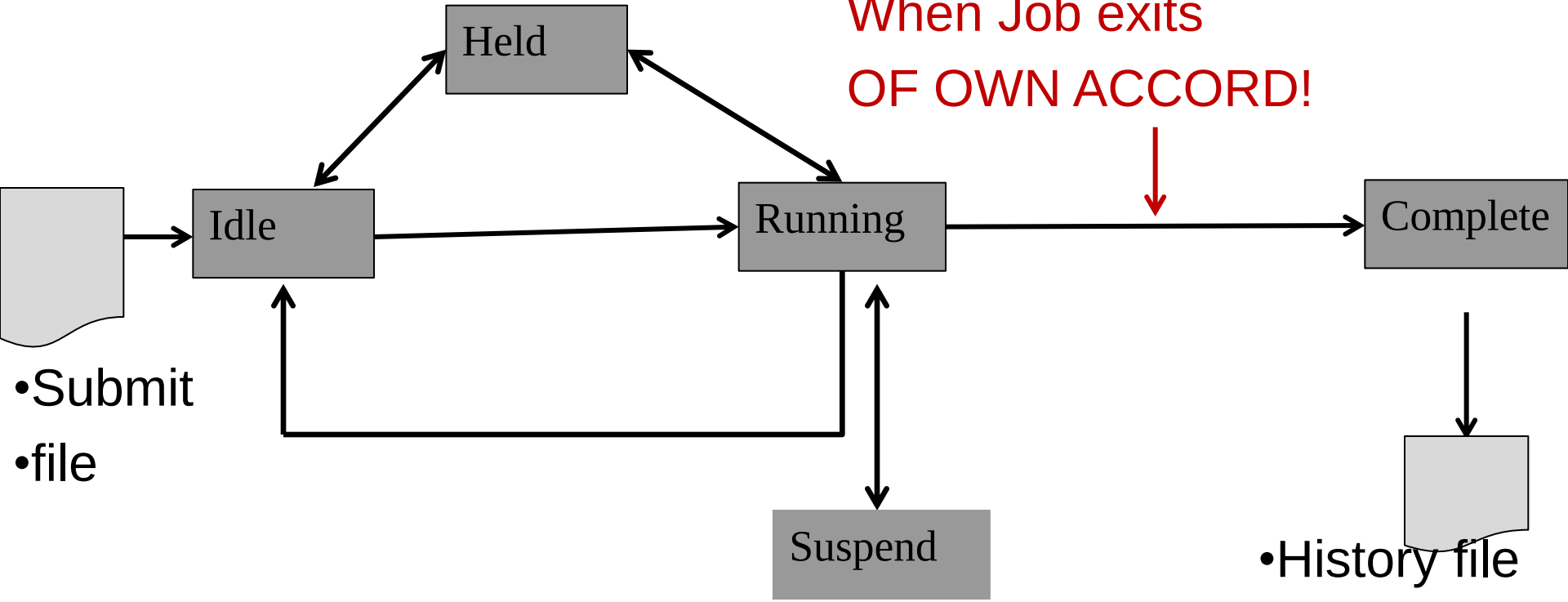
Life cycle of HTCondor Job



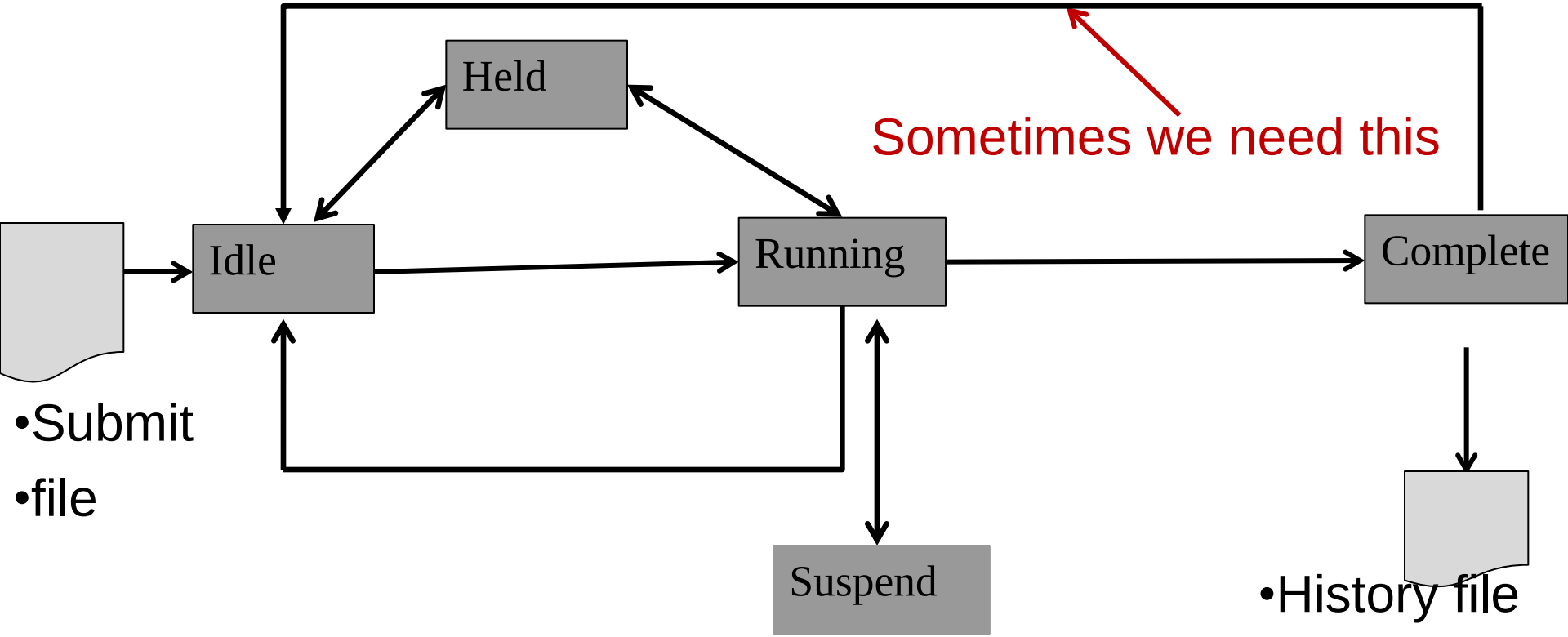
Life cycle of HTCondor Job



Life cycle of HTCondor Job



Life cycle of HTCondor Job



ON_EXIT_REMOVE

Default: true

Means always remove

```
Executable = runme.sh
Output  = out
Error   = err
Log     = log
Request_Memory = 1024
ON_EXIT_REMOVE = true
queue
```

ON_EXIT_REMOVE

False

Means never remove

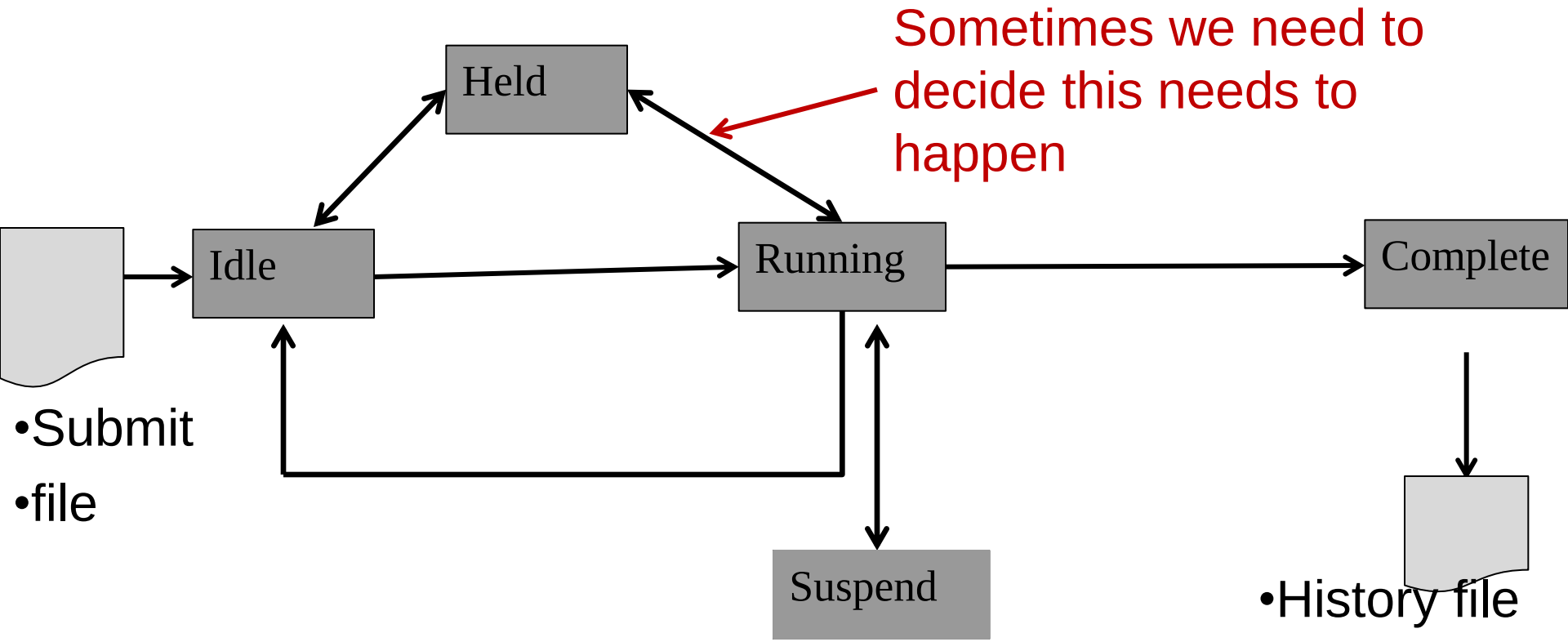
(Don’t ever do this)

```
Executable = runme.sh
Output  = out
Error   = err
Log     = log
Request_Memory = 1024
ON_EXIT_REMOVE = false
queue
```

ON_EXIT_REMOVE

```
Executable = runme.sh
Output  = out
Error   = err
Log     = log
Request_Memory = 1024
ON_EXIT_REMOVE = (ExitCode =?= 0) &&
     (ExitBySignal =?= false)
queue
```

Life cycle of HTCondor Job

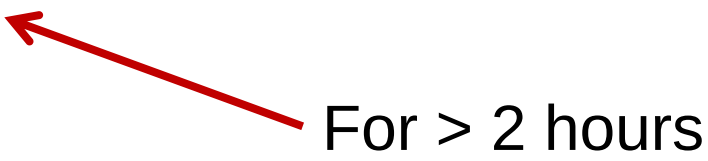


PERIODIC_HOLD

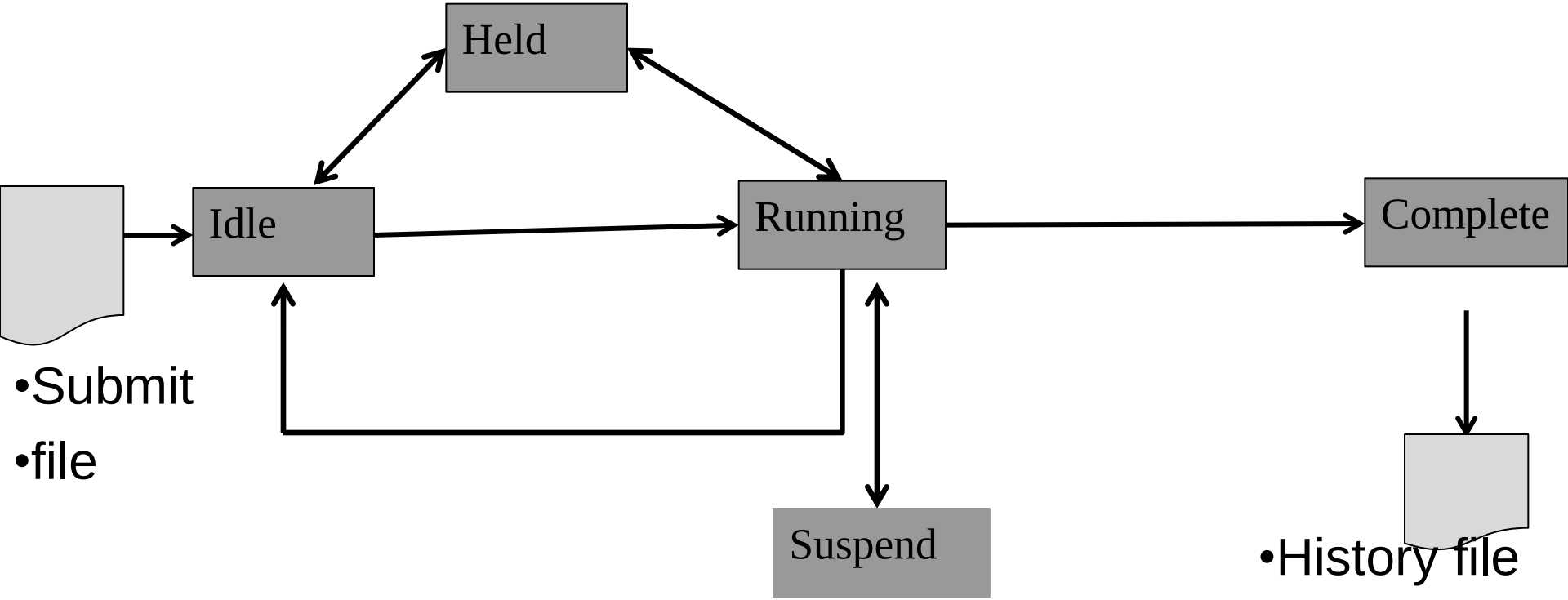
```
Executable = runme.sh
Output  = out
Error   = err
Log     = log
Request_Memory = 1024
PERIODIC_HOLD = (JobStatus == 2) &&
((CurrentTime - EnteredCurrentStatus) > (60 *
60 * 2))
queue
```

PERIODIC_HOLD

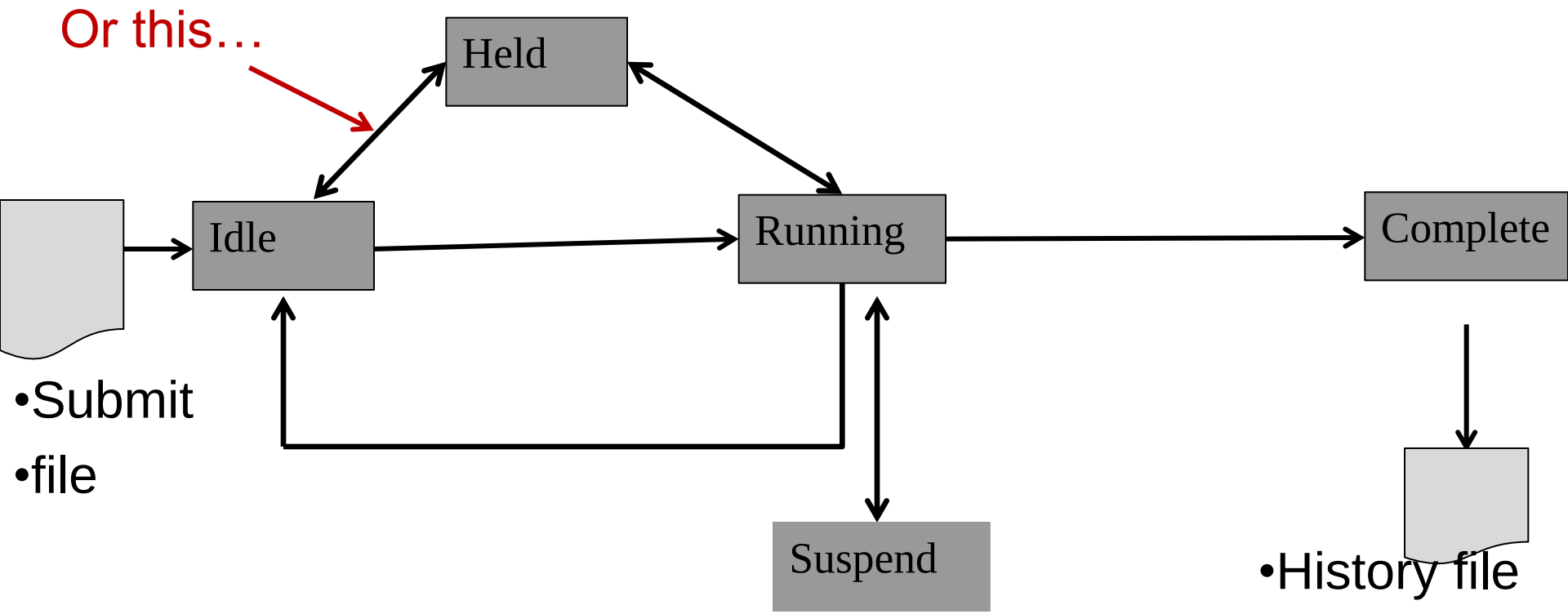
```
Executable = runme.sh
Output  = out
Error   = err
Log     = log
Request_Memory = 1024
PERIODIC_HOLD = (JobStatus == 2) &&
((CurrentTime - EnteredCurrentStatus) > (60 *
60 * 2))
queue
```



Life cycle of HTCondor Job

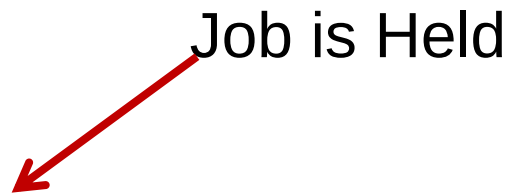


Life cycle of HTCondor Job



PERIODIC_RELEASE

```
Executable = runme.sh
Output  = out
Error   = err
Log     = log
Request_Memory = 1024
periodic_release = (JobStatus == 5) &&
   (HoldReason == 3) && (NumJobStarts < 5)
queue
```



Manual has hold codes

Integer Code	*Reason for Hold*	*HoldReasonSubCode*
1	The user put the job on hold with *condor_hold*.	
2	Globus middleware reported an error.	The GRAM error number.
3	The `PERIODIC_HOLD` expression evaluated to `True`.	
4	The credentials for the job are invalid.	
5	A job policy expression evaluated to `Undefined`.	
6	The *condor_starter* failed to start the executable.	The Unix errno number.
7	The standard output file for the job could not be opened.	The Unix errno number.
8	The standard input file for the job could not be opened.	The Unix errno number.
9	The standard output stream for the job could not be opened.	The Unix errno number.
10	The standard input stream for the job could not be opened.	The Unix errno number.
11	An internal HTCondor protocol error was encountered when transferring files.	
12	The *condor_starter* or *condor_shadow* failed to receive or write job files.	The Unix errno number.
13	The *condor_starter* or *condor_shadow* failed to read or send job files.	The Unix errno number.
14	The initial working directory of the job cannot be accessed.	The Unix errno number.
15	The user requested the job be submitted on hold.	
16	Input files are being spooled.	
17	A standard universe job is not compatible with the *condor_shadow* version available on the submitting machine.	

PERIODIC_REMOVE

```
Executable = runme.sh
Output  = out
Error   = err
Log     = log
Request_Memory = 1024
Periodic_remove = (NumJobStarts > 5)
queue
```

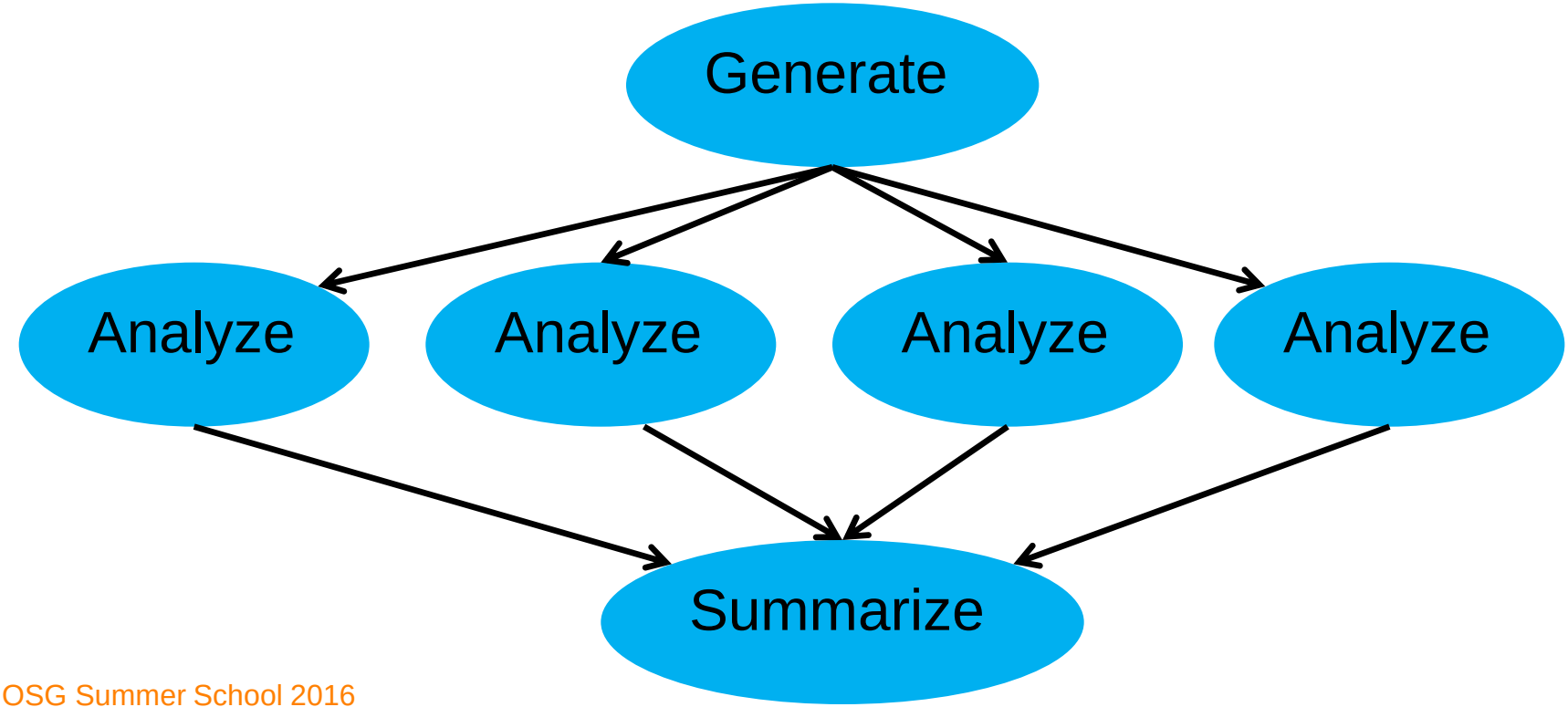
We apologize for this slide…

```
Executable = runme.sh
…
request_memory = ifthenelse(MemoryUsage =!=    undefined,
max({2048,(MemoryUsage * 3/2)}), 2048)
periodic_hold = (MemoryUsage >= ((RequestMemory) * 5/4 )) &&
       (JobStatus == 2)
periodic_release = (JobStatus == 5) &&
       ((CurrentTime - EnteredCurrentStatus) > 180) &&
       (NumJobStarts < 5) && (HoldReasonCode =!= 13) &&
       (HoldReasonCode =!= 34)
queue
```

Workflows

- Often, you don't have independent tasks!
- Common example:
  - You want to analyze a set of images
    1. You need to generate N images (once)
    2. You need to analyze all N images
       - One job per image
    3. You need to summarize all results (once)

Do you want to do this manually?

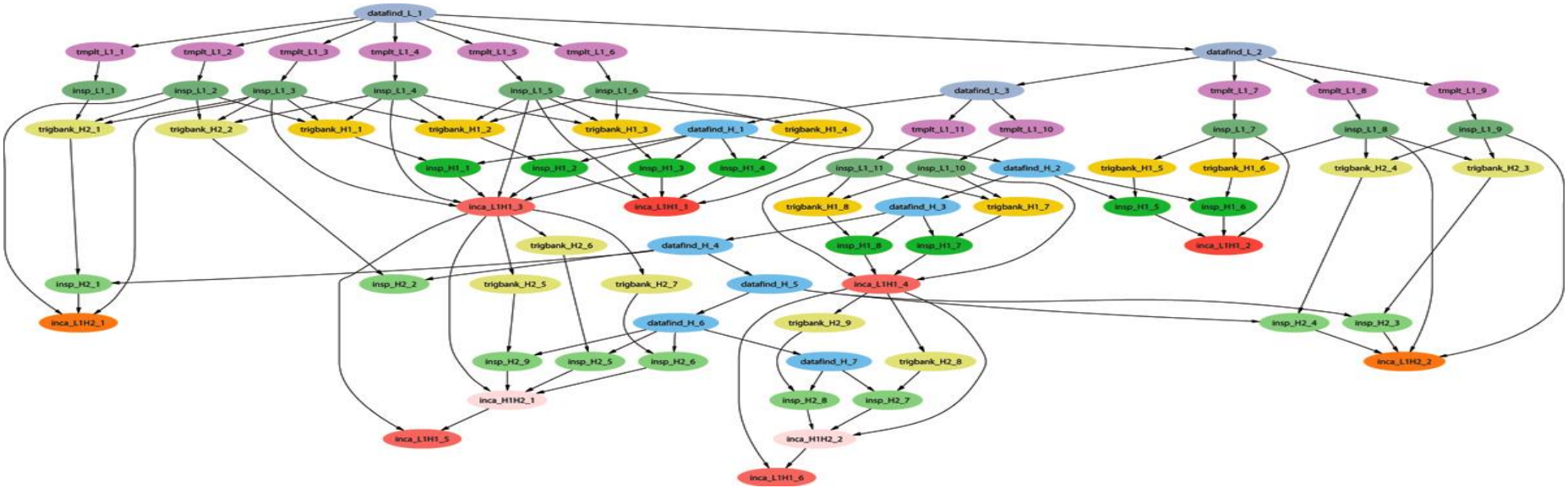


Workflows: The HTC definition

Workflow:

A graph of jobs to run: one or more jobs must **succeed** before one or more others can start running

Example of a LIGO Inspiral DAG



DAGMan

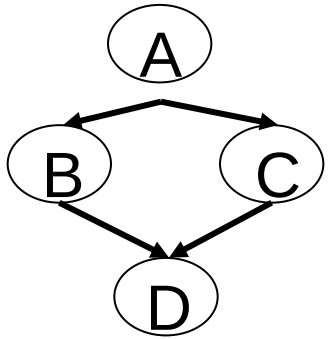
- DAGMan: HTCondor's workflow manager
  - Directed Acyclic Graph (DAG)
  - Manager (Man)
- Allows you to specify the dependencies between your HTCondor jobs
- Manages the jobs and their dependencies

- That is, it manages a workflow of HTCondor jobs

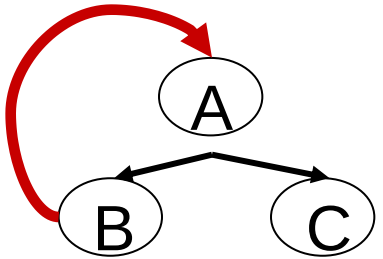
What is a DAG?

- A DAG is the structure used by DAGMan to represent these dependencies.
- Each job is in a node in the DAG.
- Each node can have any number of “parent” or “children” nodes – as long as there are no loops!

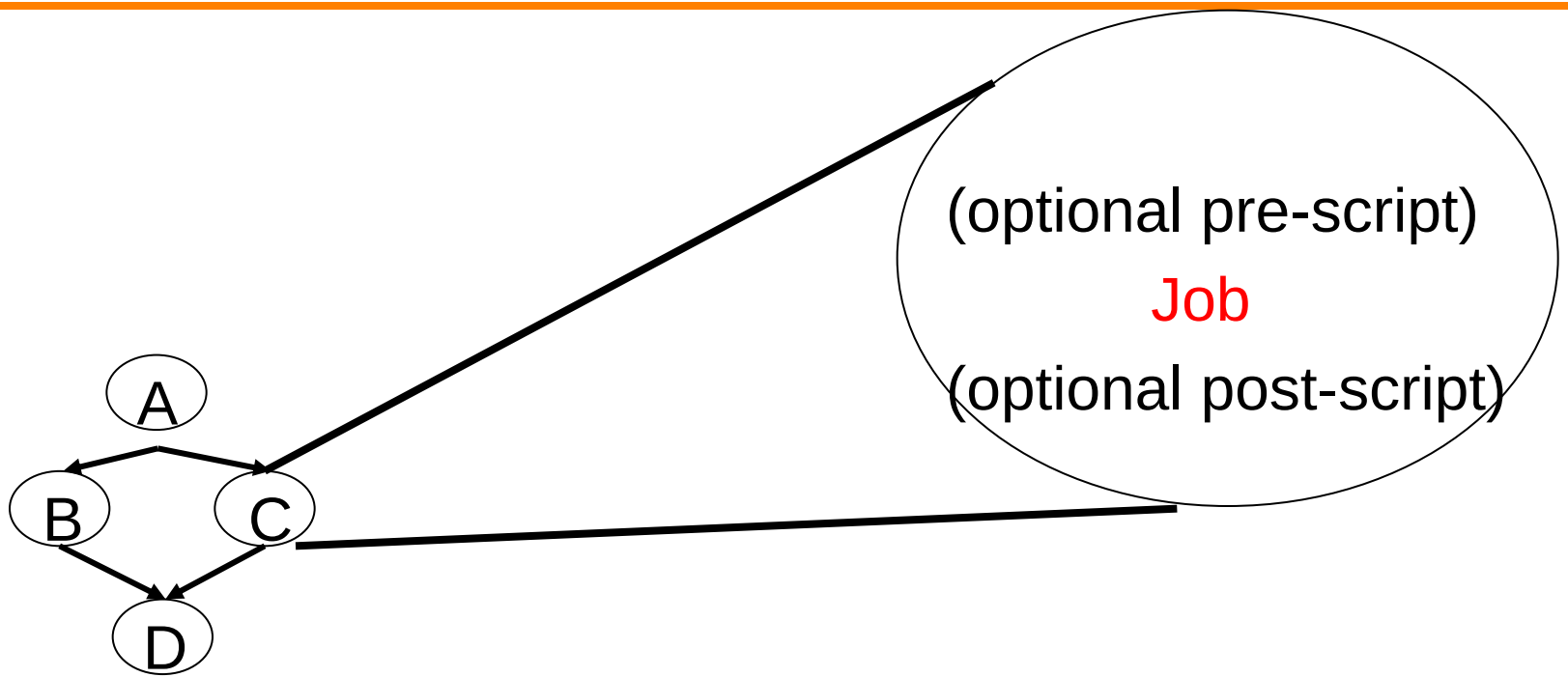
OK:



Not OK:



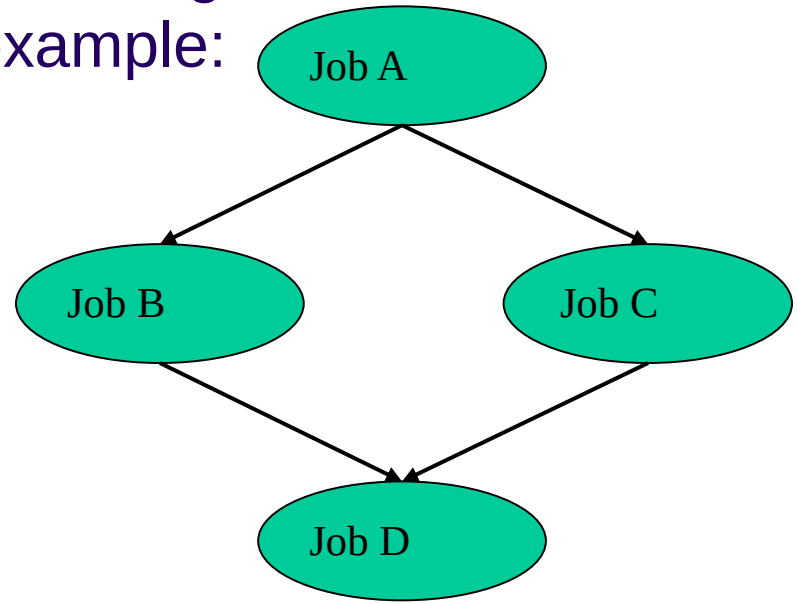
So, what's in a node?



Defining a DAG

- A DAG is defined by a *.dag file*, listing each of its nodes and their dependencies. For example:

```
# Comments are good
Job A a.sub
Job B b.sub
Job C c.sub
Job D d.sub

Parent A Child B C
Parent B C Child D
```



DAG Files….

- This complete DAG has five htcondor files

One DAG File:

```
Job A a.sub
Job B b.sub
Job C c.sub
Job D d.sub

Parent A Child B C
Parent B C Child D
```

Four Submit Files:

```
Universe = Vanilla
Executable = analysis…
```

```
Universe = …
```

Submitting a DAG

- To start your DAG, just run **`condor_submit_dag`** with your .dag file, and HTCondor will start a DAGMan process to manage your jobs:

```
% condor_submit_dag diamond.dag
```

- condor_submit_dag submits a Scheduler Universe job with DAGMan as the executable
- Thus the DAGMan daemon itself runs as an HTCondor job, so you don’t have to baby-sit it

DAGMan is a HTCondor job

- DAGMan itself is a condor job with a job id, so

```
% condor_rm job_id_of_dagman
% condor_hold job_id_of_dagman
% condor_q -dag # is  magic
```

- DAGMan submits jobs, one cluster per node
- Don’t confuse dagman as job with jobs of dagman

Some examples:

```
$ condor_submit_dag test.dag
File for submitting this DAG to HTCondor: test.dag.condor.sub
Log of DAGMan debugging messages: test.dag.dagman.out
Log of HTCondor library output  : test.dag.lib.out
Log of HTCondor library error messages: test.dag.lib.err
Log of the life of condor_dagman itself: test.dag.dagman.log

Submitting job(s).
1 job(s) submitted to cluster 64.
```

Some examples:

```
$ condor_q
-- Schedd: learn.chtc.wisc.edu : <128.104.100.43:9618?...
 ID      OWNER            SUBMITTED     RUN_TIME ST PRI SIZE CMD
  64.0   gthain          7/19 11:03   0+00:00:03 R  0    0.3
condor_dagman -p 0 -f -l . -Lockfile test.dag.lock -AutoRe

1 jobs; 0 completed, 0 removed, 0 idle, 1 running, 0 held, 0
suspended
```

Some examples:

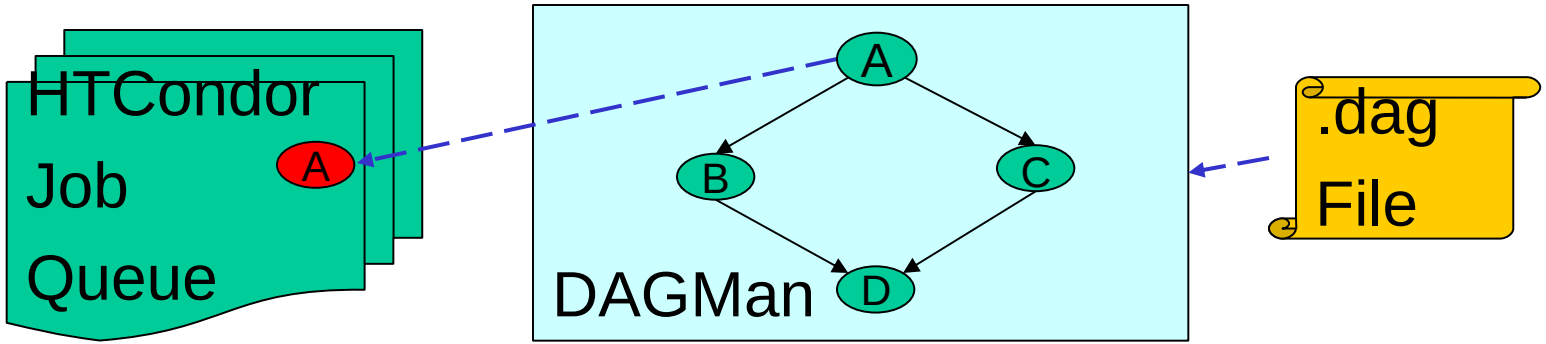
```
$ condor_q -dag
-- Schedd: learn.chtc.wisc.edu : <128.104.100.43:9618?...
 ID      OWNER/NODENAME      SUBMITTED     RUN_TIME ST PRI SIZE
64.0   gthain             7/19 11:03   0+00:00:07 R  0    0.3
condor_dagman -p 0 -f -l . -Lockfile test.dag.lock -Aut
  65.0    |-A               7/19 11:04   0+00:00:00 I  0    0.0
2 jobs; 0 completed, 0 removed, 1 idle, 1 running, 0 held, 0
suspended
```

Some examples:

```
$ condor_submit_dag test.dag

ERROR: "test.dag.condor.sub" already exists.
ERROR: "test.dag.lib.out" already exists.
ERROR: "test.dag.lib.err" already exists.
ERROR: "test.dag.dagman.log" already exists.

Some file(s) needed by condor_dagman already exist.  Either
rename them,
use the "-f" option to force them to be overwritten, or use
the "-update_submit" option to update the submit file and
continue.
```

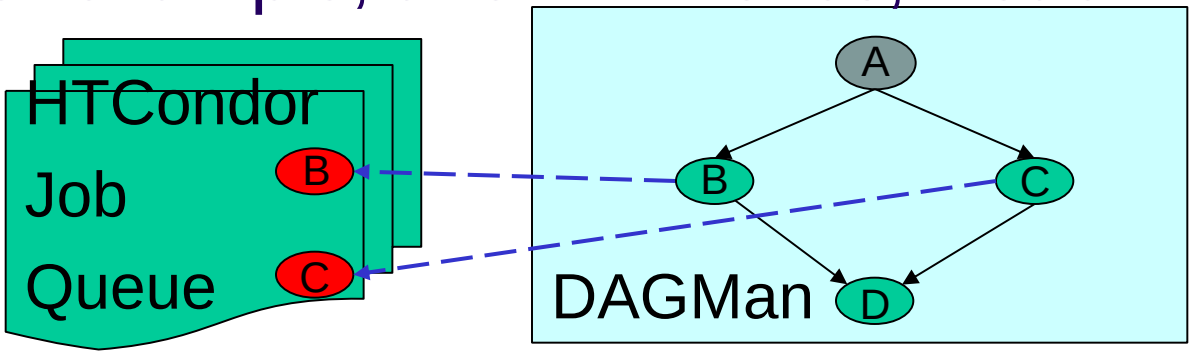
Running a DAG

- DAGMan acts as a job scheduler, managing the submission of your jobs to HTCondor based on the DAG dependencies



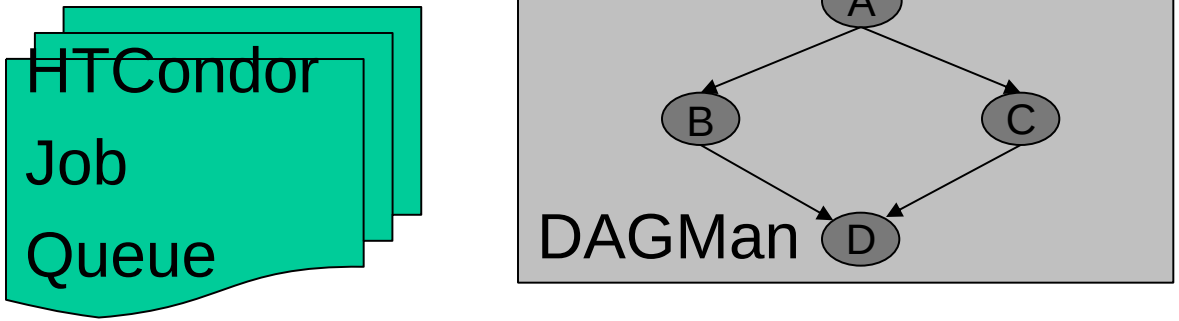
Running a DAG (cont'd)

- DAGMan submits jobs to HTCondor at the appropriate times
- For example, after A finishes, it submits B & C



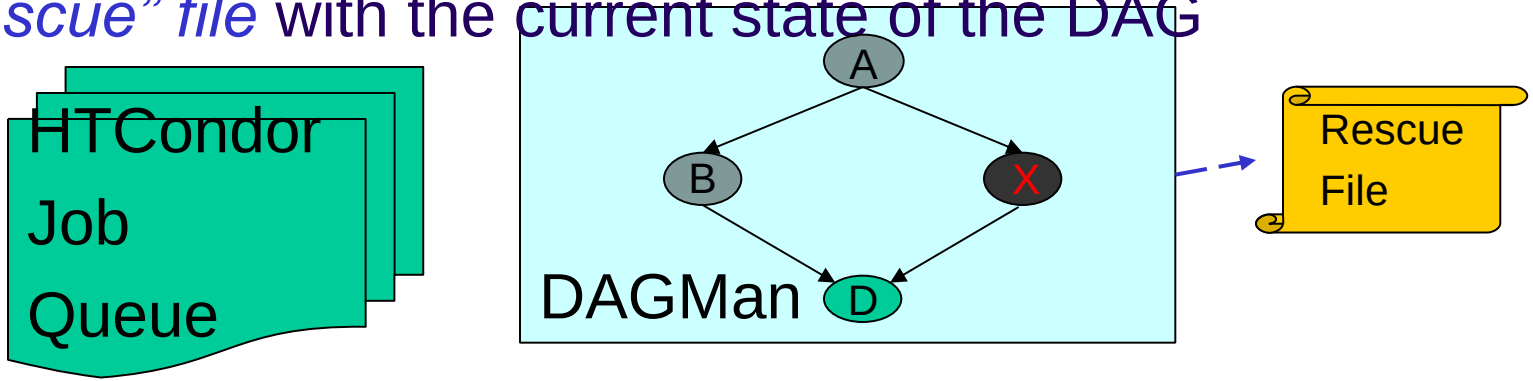
Finishing a DAG

- Once the DAG is complete, the DAGMan job itself is finished, and exits



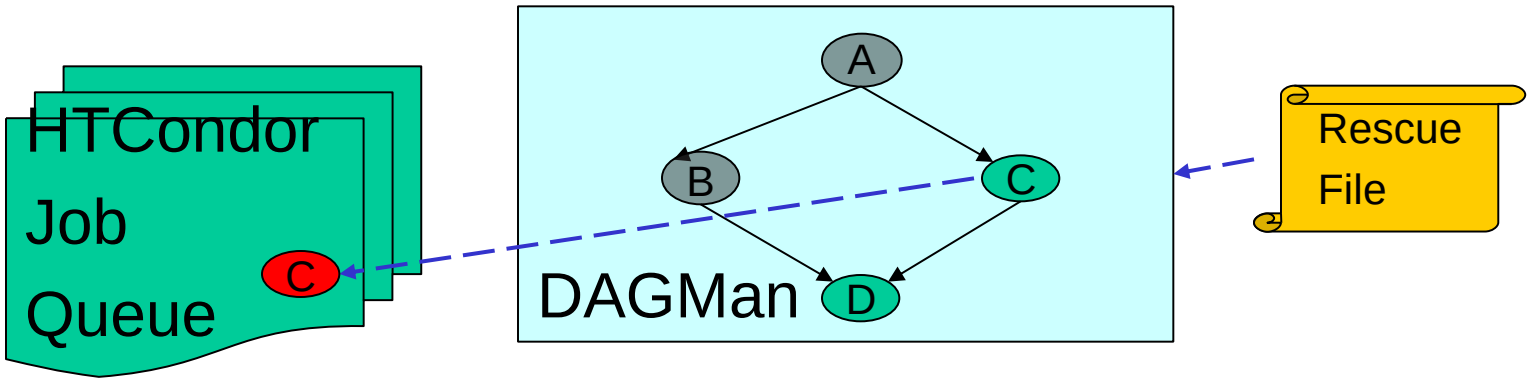
What if a job fails?

- A job *fails* if it exits with a non-zero exit code
- In case of a job failure, DAGMan runs other jobs until it can no longer make progress, and then creates a *"rescue" file* with the current state of the DAG



Recovering a DAG

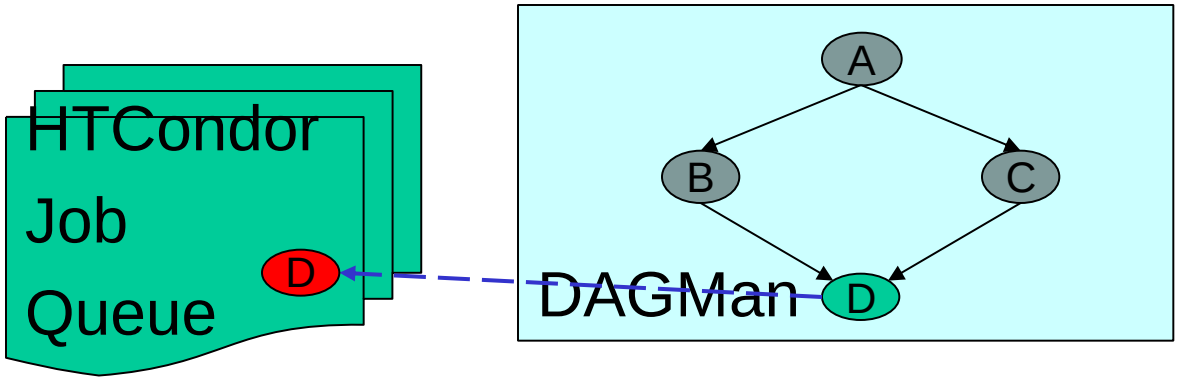
- Once the failed job is ready to be re-run, the rescue file can be used to restore the prior state of the DAG



Another example of reliability for HTC!

Recovering a DAG (cont'd)

- Once that job completes, DAGMan will continue the DAG as if the failure never happened



DAGMan & Fancy Features

- DAGMan doesn't have a lot of "fancy features"
  - No loops
  - No help to make very large DAGs
- Focus is on solid core
  - Add the features people need in order to run large DAGs well
  - People build systems on top of DAGMan

Related Software

Pegasus: http://pegasus.isi.edu/

- Writes DAGs based on abstract description
- Runs DAG on appropriate resource (HTCondor, OSG, EC2…)
- Locates data, coordinates execution
- Uses DAGMan, works with large workflows

Makeflow: http://nd.edu/~ccl/software/makeflow/

- User writes *make* file, not DAG
- Handles data transfers to remote systems

DAGMan: Reliability

- For each job, HTCondor generates a log file
- DAGMan reads this log to see what has happened
- If DAGMan dies (crash, power failure, etc…)
  - HTCondor will restart DAGMan
  - DAGMan re-reads log file
  - DAGMan knows everything it needs to know
  - Principle: DAGMan can recover state from files and without relying on a service (HTCondor queue, database…)
- Recall: HTC requires reliability!

Let’s try it out!

- Exercises with DAGMan.

Questions?

- Questions? Comments?
- Feel free to ask me questions later: