Checkpointing on OSPool

Showmic Islam

Research Computing Facilitator@ OSG

HPC Application Specialist

Holland Computing Center

University of Nebraska-Lincoln

Outline

What?

What is checkpointing?
What jobs are suitable for checkpointing?

Why?

Why checkpointing is needed?

How?

How to checkpoint?
Different methods for checkpointing

What?

PATh

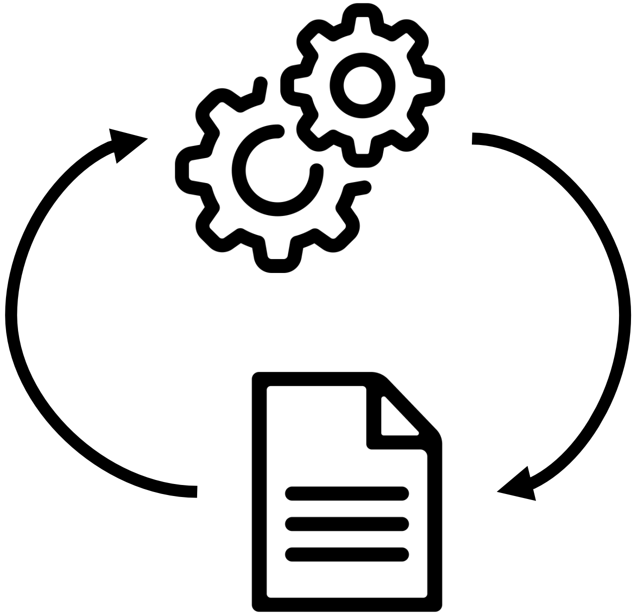
What is Checkpointing?

- **According to ChatGPT-** Checkpointing is a technique to save the state of a computation so that it can be resumed later without losing progress.
- Analogy: Saving progress in a game periodically
- The executable periodically saves its progress to disk – a *self*-made *checkpoint* – so that it can resume from that point if interrupted later, losing minimal progress

Save game from the Noun Project

Requirement of Jobs

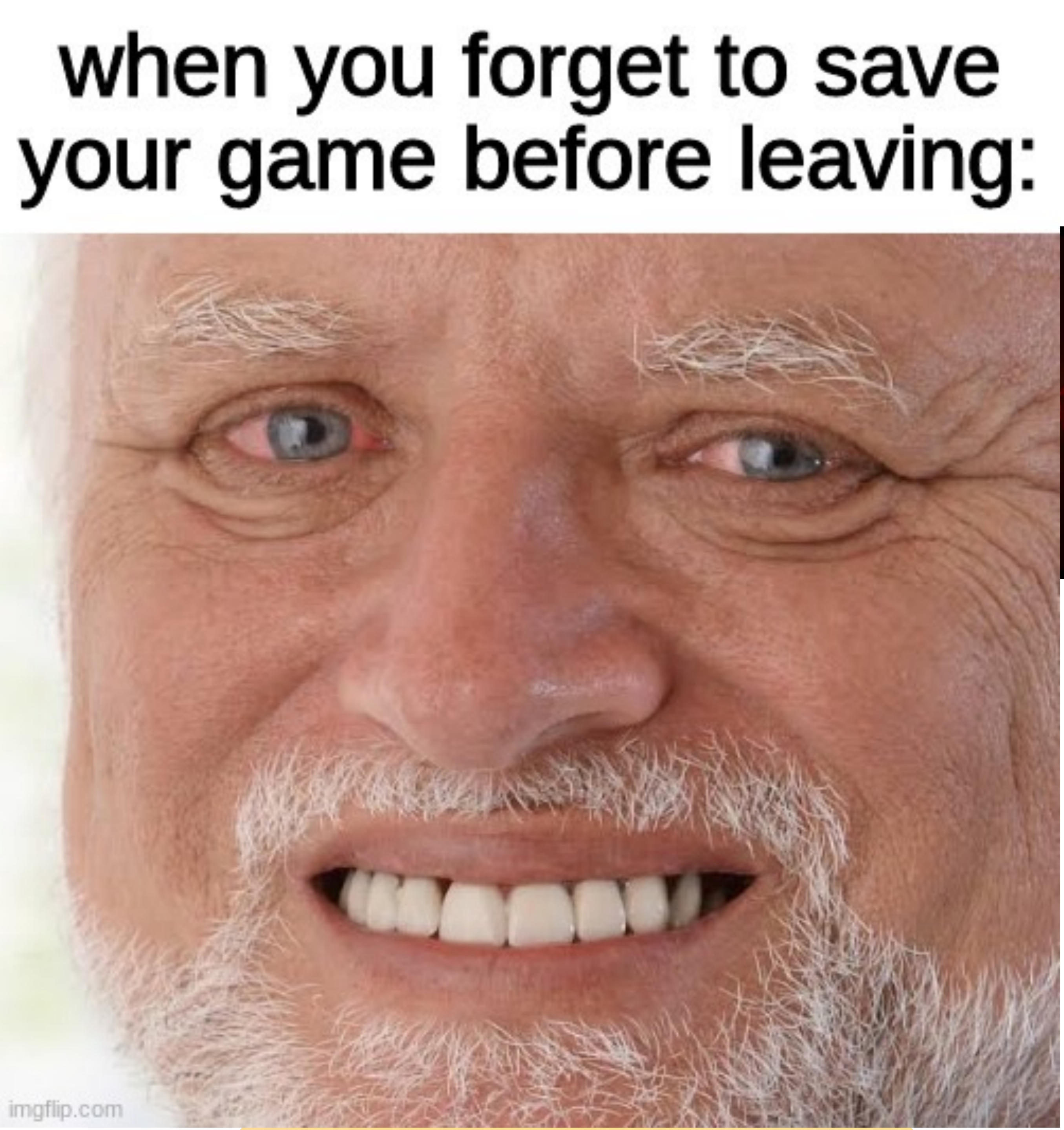
- **Ability to checkpoint and restart:**
  - *Checkpoint*: Periodically write state to a file on disk.
  - *Restart*: Code can both find the checkpoint file and can resume from it.
  - *Exit*: Code exits with a non-zero exit code after writing a certain number of checkpoints, exits normally after writing final output.
  - (May need a wrapper script to do some of this.)
- **Ability to checkpoint sufficiently* frequently**



File by Tanvir Islam from the Noun Project
Gears by Gregor Cresnar from the Noun Project

* Varies by code and available resources

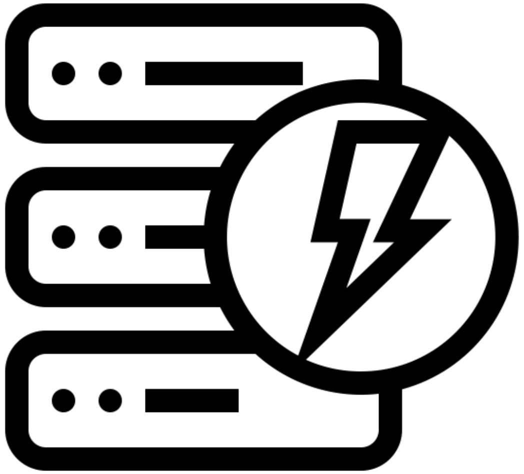
Why?



when you forget to save your game before leaving:

Why to Checkpoint

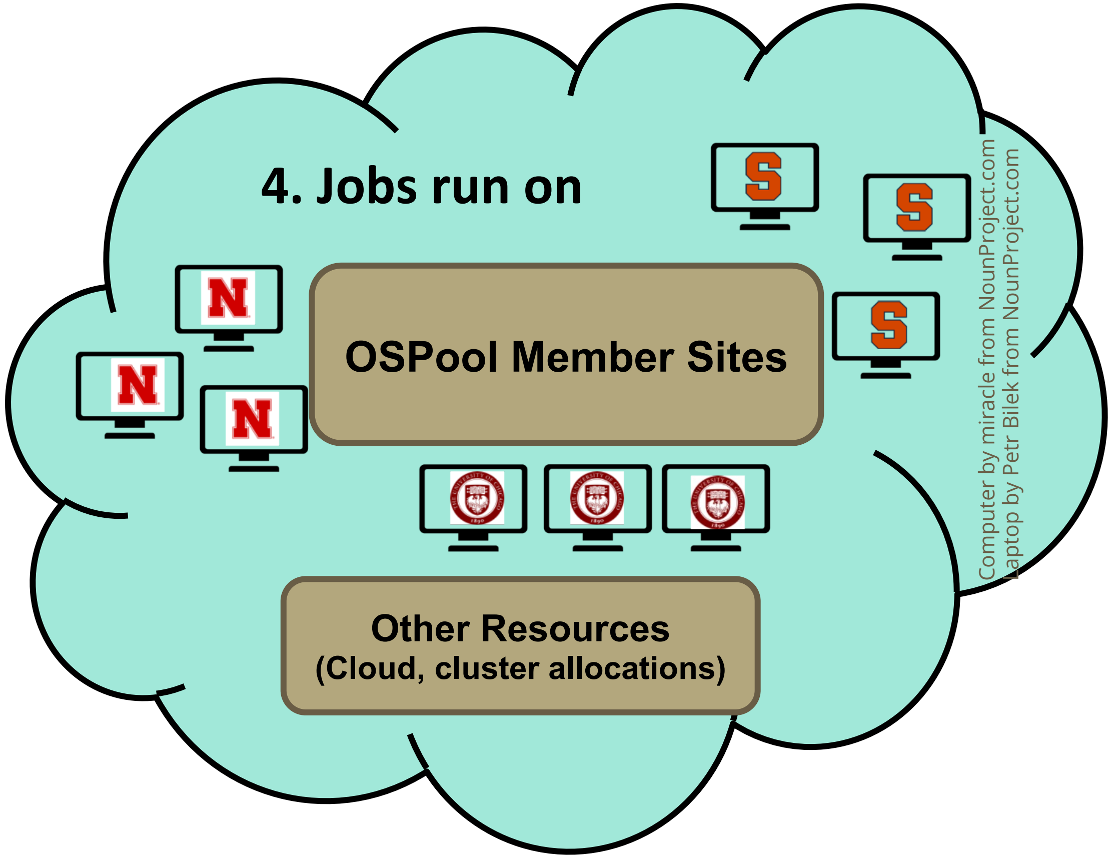
- **Interruptions happen**:
  - Hardware or networking failures
  - Cluster/node policy (jobs can only run for 8 hours before getting killed)
  - Using opportunistic or backfill resources with no runtime guarantee
- Self-checkpointing allows you to make progress through interruptions, especially for longer-running jobs.



Lightning by Bernar Novalyi from the Noun Project

Characteristics of OSPool

- The maximum allowed job duration on the OSPool is 20 hours*
- Jobs on the OSPool runs on an opportunistic manner
- The longer a job runs on OSPool the greater the probability that your job may get interrupted
- Checkpointing removes the wall-time limit on the OSPool
- Checkpointing increases the goodput of the jobs



4. Jobs run on

OSPool Member Sites

Other Resources
(Cloud, cluster allocations)

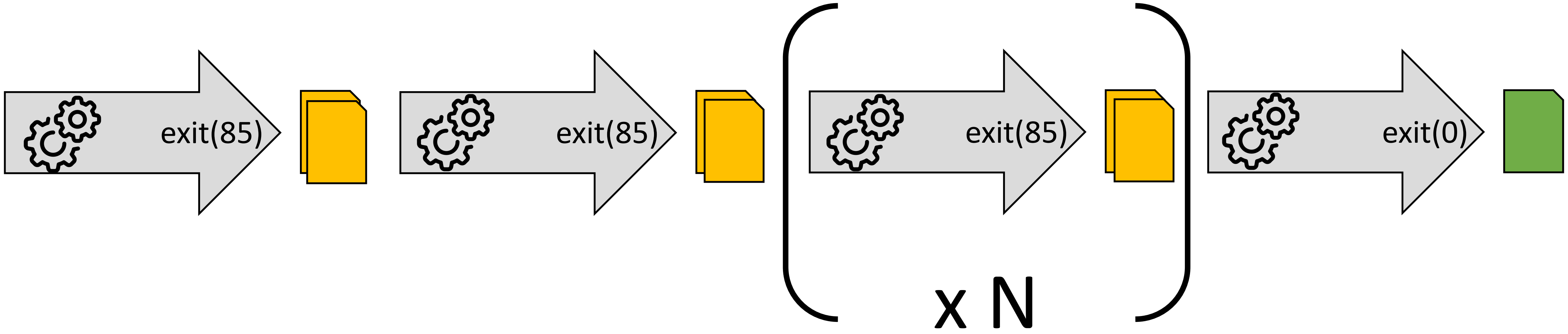
Computer by miracle from NounProject.com
Laptop by Petr Bilek from NounProject.com

How?

Ways to Checkpoint

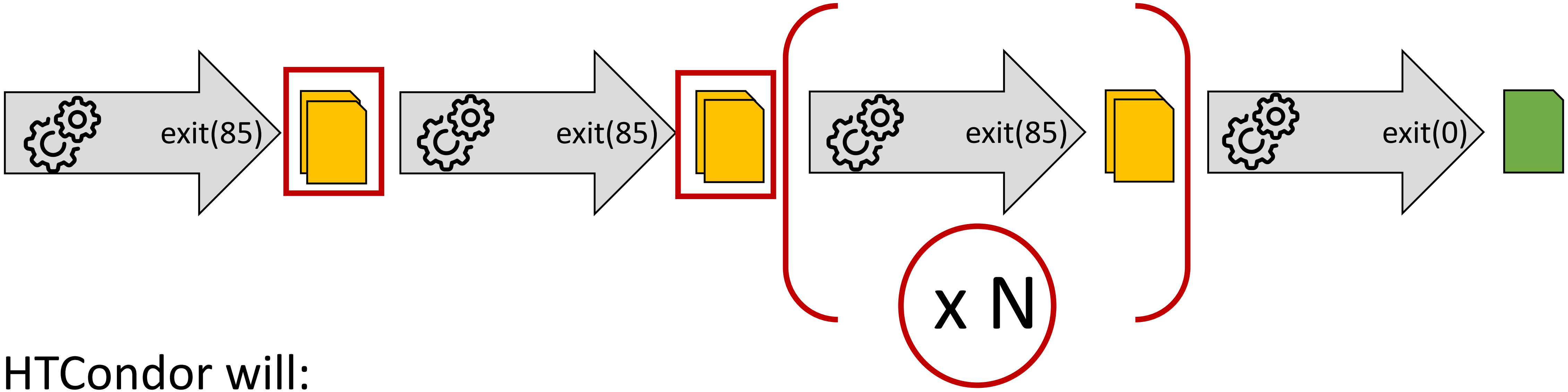
- **Exit-driven self-checkpointing**
  - Since HTCondor ≥ 8.9.7
  - *Waaaay* better for most use cases, esp. in OSG
  - What is shown here
- Eviction-driven self-checkpointing
  - Not even worth talking about for OSG!
  - Documented in the HTCondor Manual
  - But don't use it 😁

Executable Exits After Checkpoint



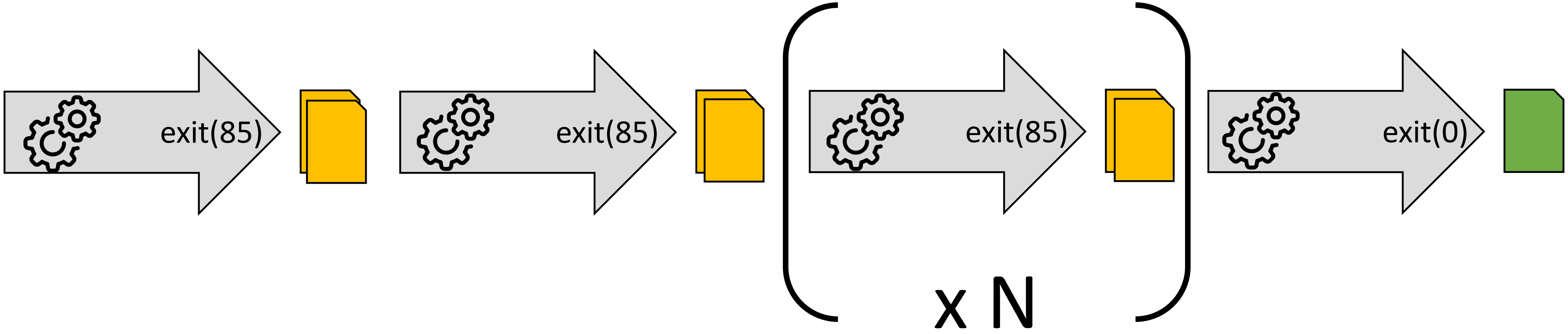
- Each executable run:
  - Produces checkpoint file(s)
  - Exits with a specific code when checkpointing, and a final exit code when done.
- Note that the executable, on its own, won't run a complete execution. It needs an external process to make it repeat.

Save Checkpoint File/Resume with HTCondor



- HTCondor will:
  - Restart the executable until the overall calculation is done (exit 0).
  - Copy the checkpoint file(s) to a persistent location, to facilitate restarts if the job is interrupted.

Save Checkpoint File/Resume with HTCondor



executable =

checkpoint_exit_code = 85

transfer_checkpoint_files =

Example Submit file

```
executable = my_software

transfer_input_files = my_input.txt
transfer_checkpoint_files = checkpoint.txt

log = example.log
error = example.err
output = example.out
transfer_output_files = my_output.txt

checkpoint_exit_code = 85


queue
```

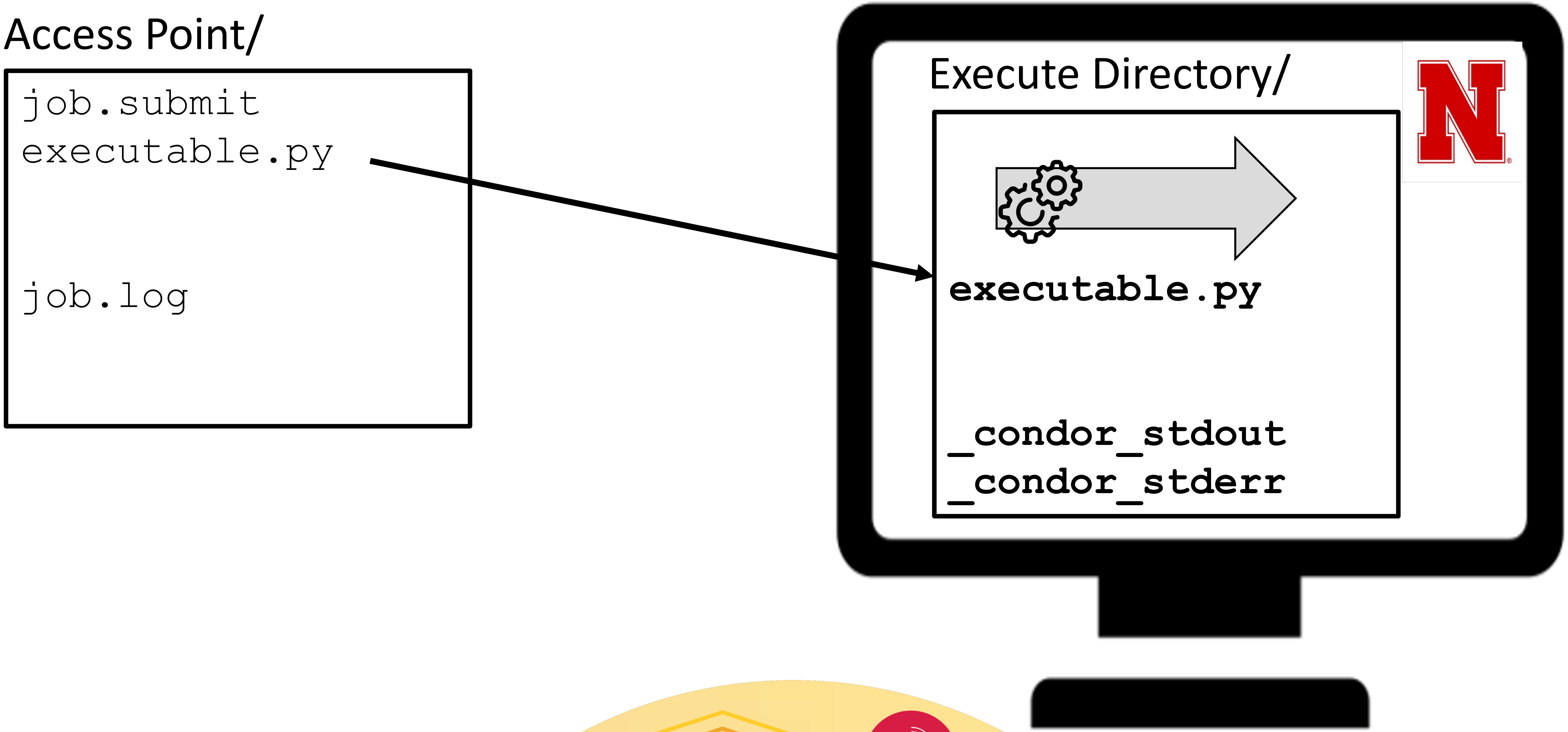
Job Submitted

Access Point/

`job.submit`
`executable.py`

`job.log`

Job Starts, Executable Starts



Access Point/

job.submit
executable.py

job.log

Execute Directory/

executable.py

_condor_stdout
_condor_stderr

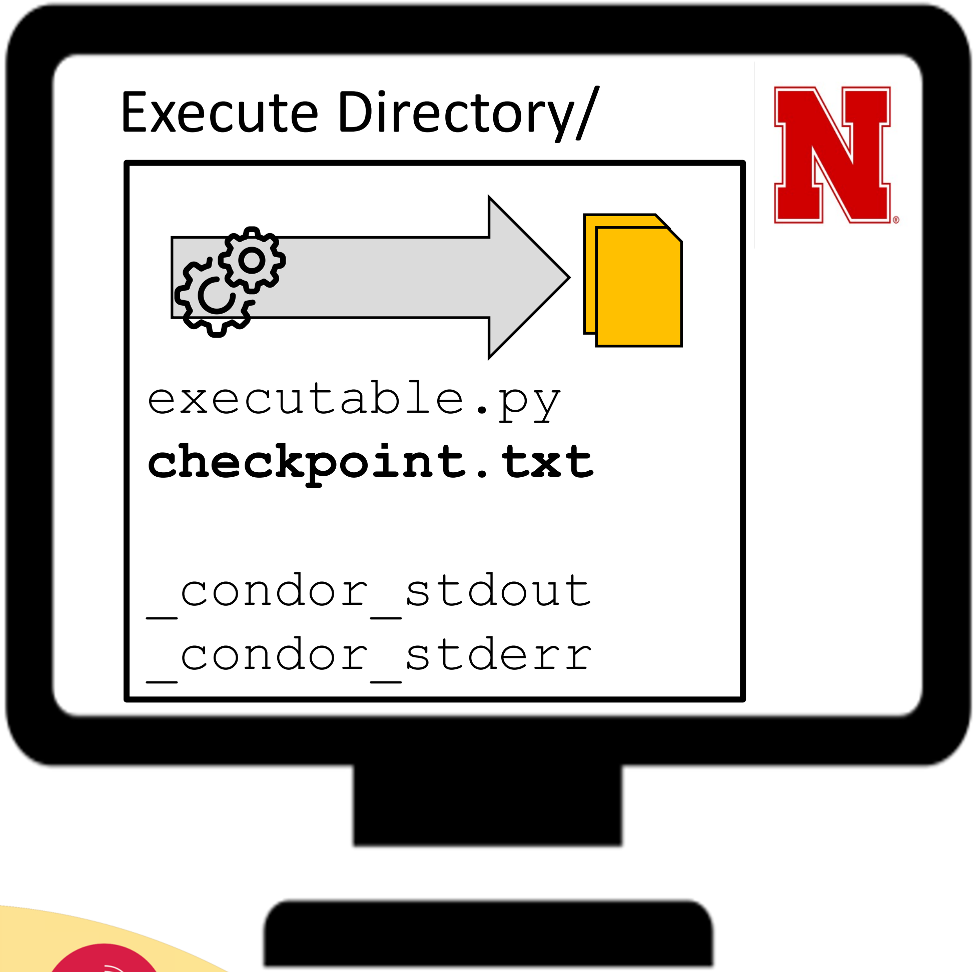
Executable Checkpoints

Access Point/

```
job.submit
executable.py



job.log
```

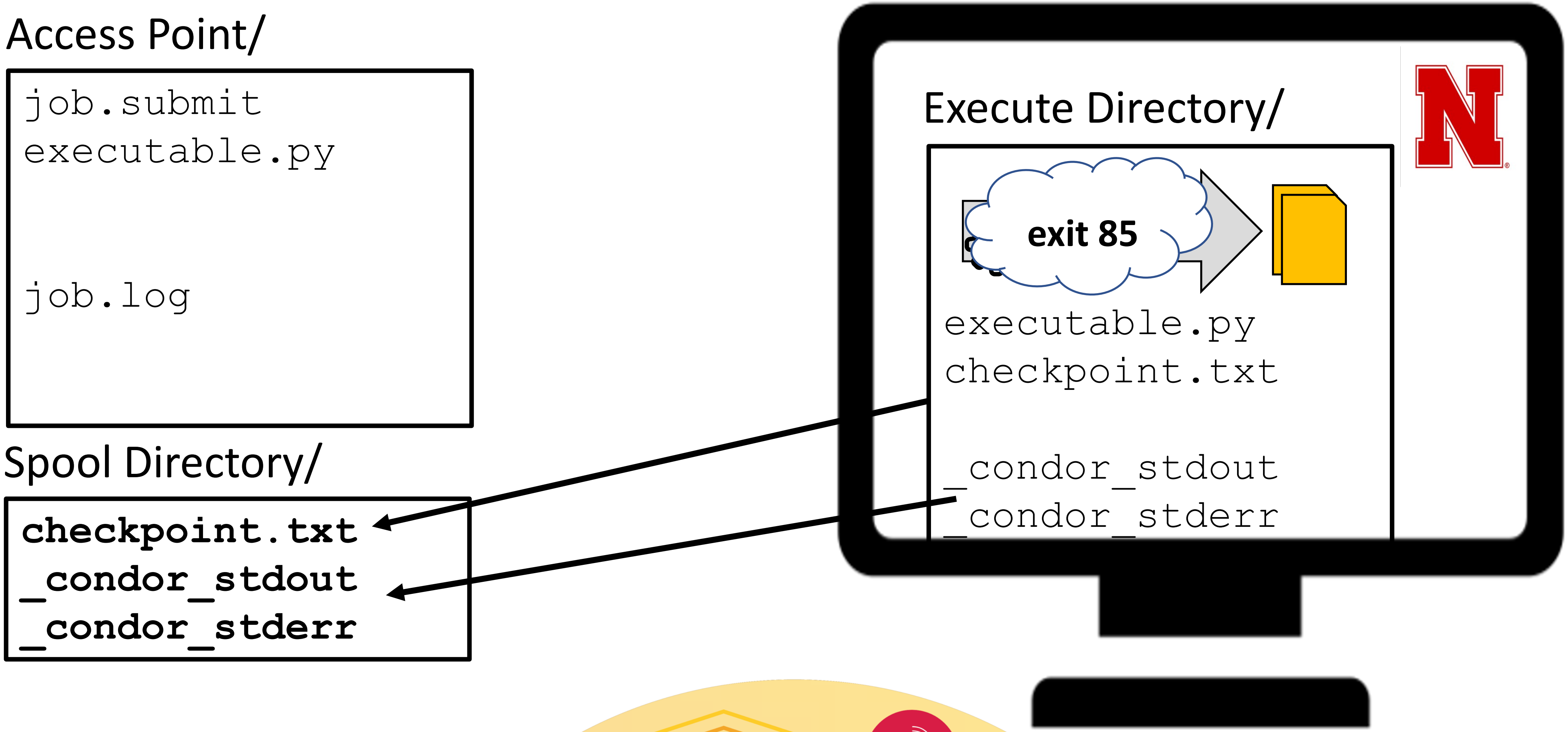


Execute Directory/

```
executable.py
checkpoint.txt

_condor_stdout
_condor_stderr
```

Executable Exits, Checkpoint Spooled



Access Point/

```
job.submit
executable.py


job.log
```

Spool Directory/

```
checkpoint.txt
_condor_stdout
_condor_stderr
```

Execute Directory/

```
executable.py
checkpoint.txt

_condor_stdout
_condor_stderr
```

Executable Started Again

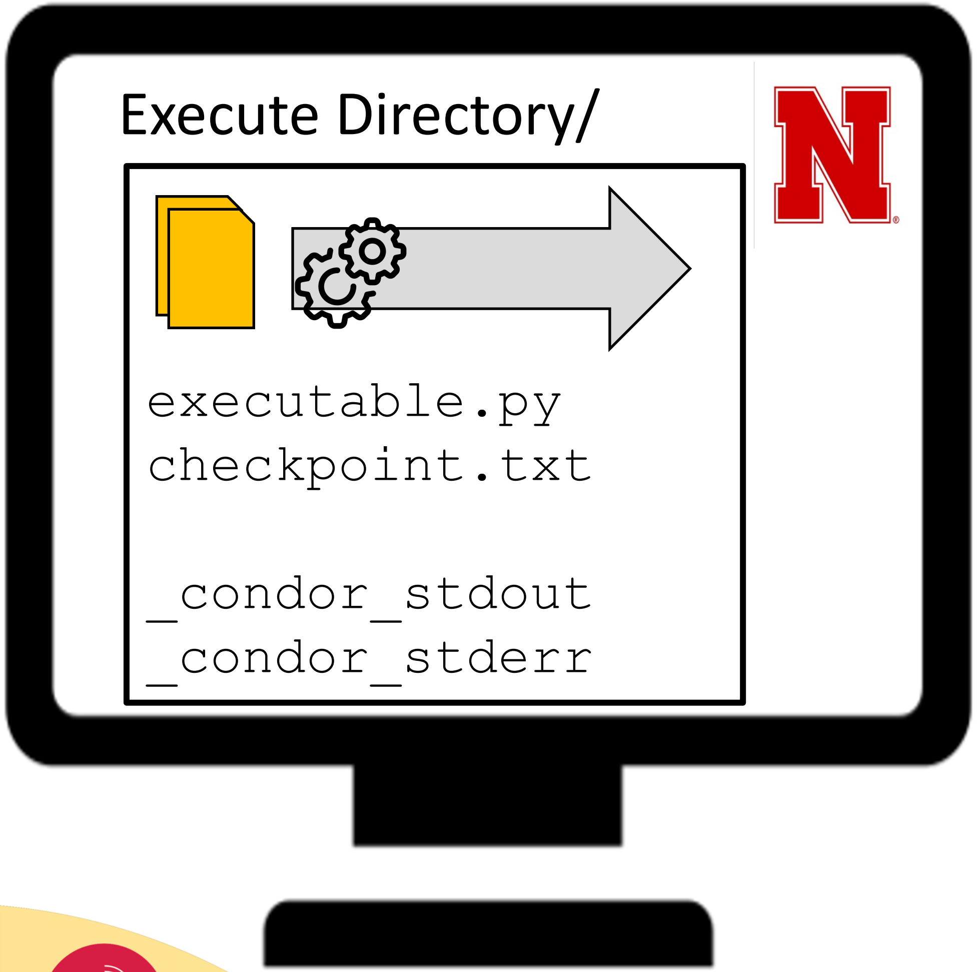
Access Point/

```
job.submit
executable.py


job.log
```

Spool Directory/

```
checkpoint.txt
_condor_stdout
_condor_stderr
```



Execute Directory/

```
executable.py
checkpoint.txt

_condor_stdout
_condor_stderr
```

Checkpoint Cycle Continues

Executable Interrupted

Access Point/

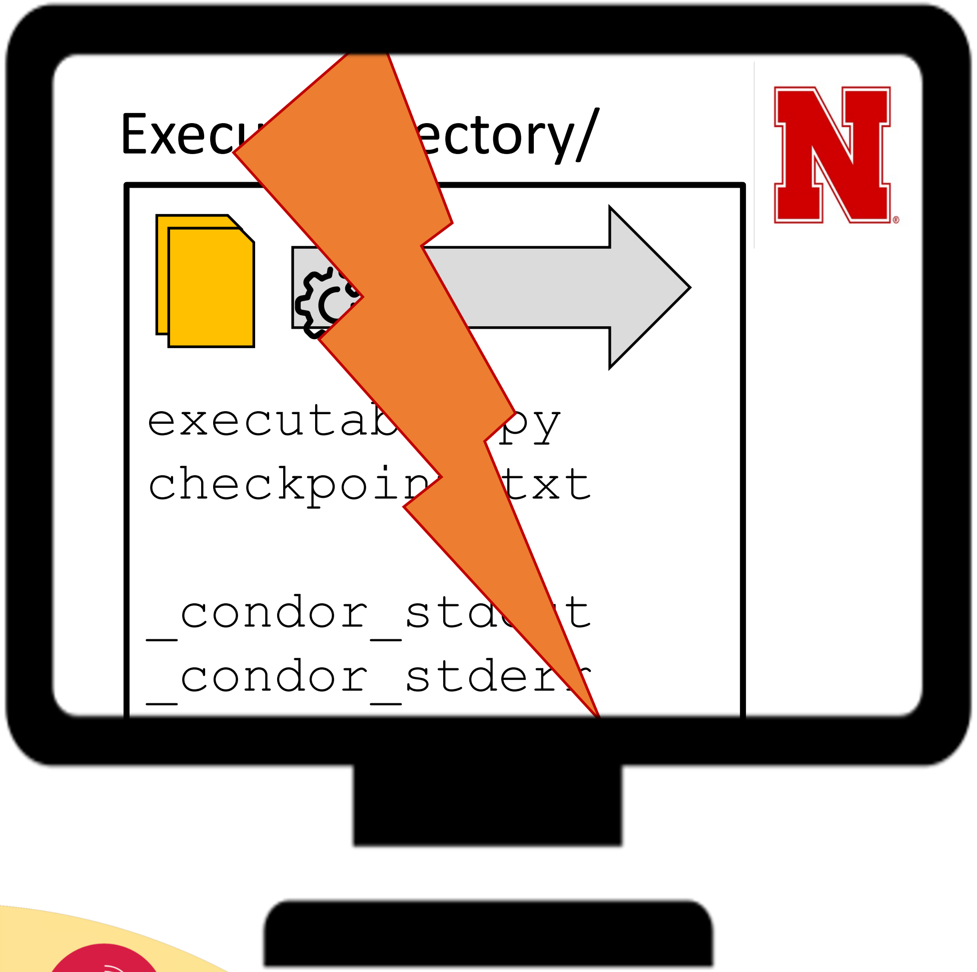
```
job.submit
executable.py



job.log
```

Spool Directory/

```
checkpoint.txt
_condor_stdout
_condor_stderr
```



Execu ectory/

```
executab  py
checkpoin  txt

_condor_std  t
_condor_stder
```

Job Idle

Access Point/

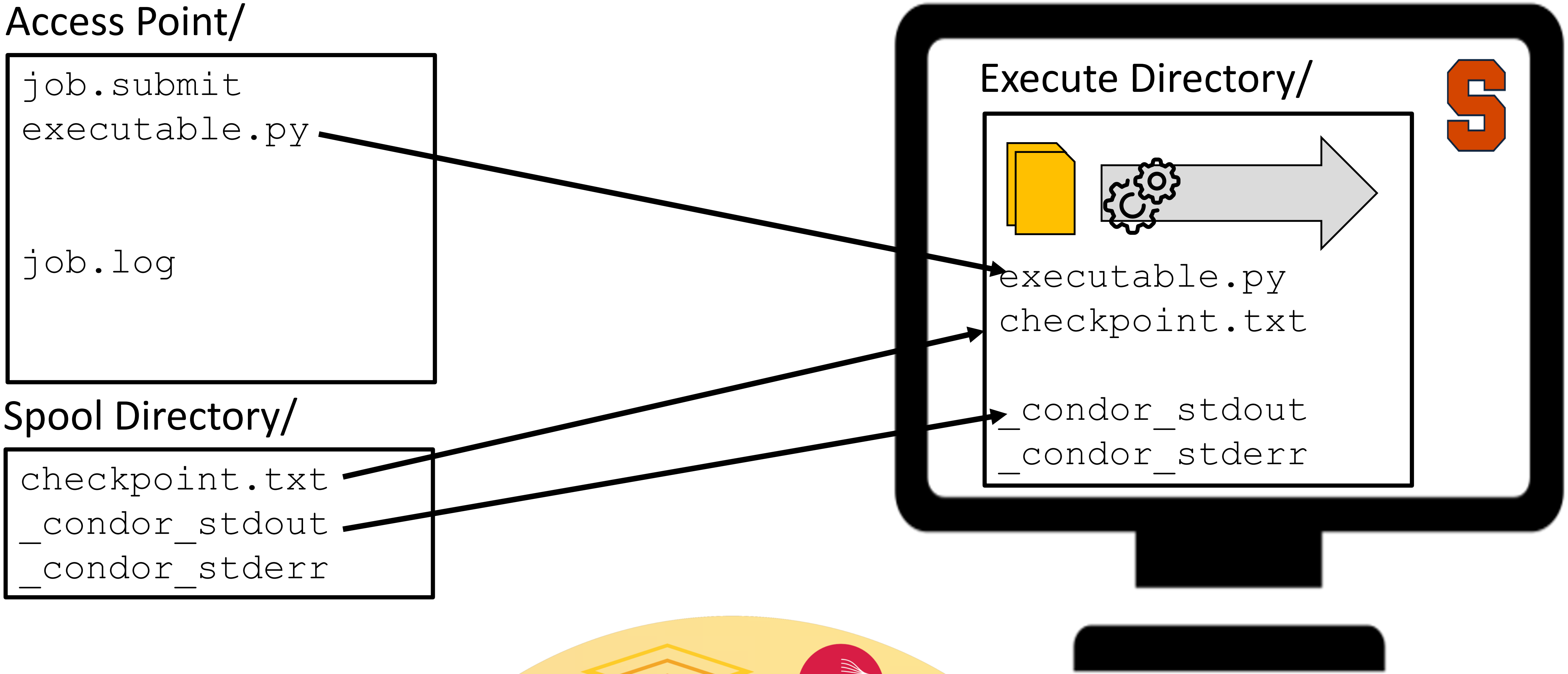
```
job.submit
executable.py


job.log
```

Spool Directory/

```
checkpoint.txt
_condor_stdout
_condor_stderr
```

Job Restarts, Executable Restarts



Access Point/

```
job.submit
executable.py


job.log
```

Spool Directory/

```
checkpoint.txt
_condor_stdout
_condor_stderr
```

Execute Directory/

```
executable.py
checkpoint.txt

_condor_stdout
_condor_stderr
```

Checkpoint Cycle Continues

Final Execution: Executable Creates Output

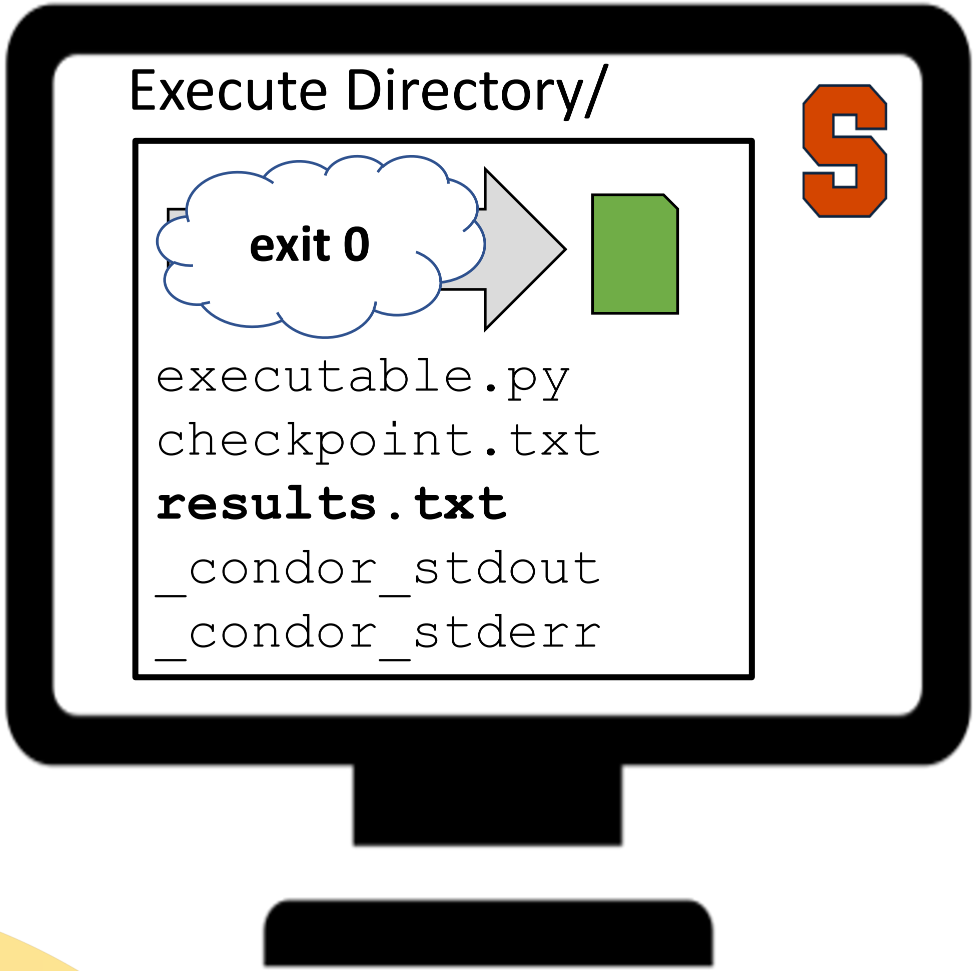
Access Point/

```
job.submit
executable.py


job.log
```

Spool Directory/

```
checkpoint.txt
_condor_stdout
_condor_stderr
```



Execute Directory/

```
executable.py
checkpoint.txt
results.txt
_condor_stdout
_condor_stderr
```

Output Returned

Access Point/

job.submit
executable.py
checkpoint.txt
results.txt
job.log
job.out
job.err

Think About Output Files

- Same mechanisms for transferring output at the end of the job (triggered by executable's exit 0)
  - New output files are transferred back to the submission directory
  - To transfer specific output files or directories, use:

```
transfer_output_files = file1, outputdir
```

- ANY output file you want to save between executable iterations (like a log file), should be included in the list of

```
transfer_checkpoint_files
```

- Older versions of HTCondor may have different default behavior

Testing and Troubleshooting

- Simulate a job interruption:
    - `condor_vacate_job` *`JobID`*
- Examine your checkpoint files in the SPOOL directory:
    - Use `condor_evicted_files` *`JobID`*
    - To find the SPOOL directory: `condor_config_val SPOOL`
- Look at the HTCondor job log for file transfer information.

Sample Code

Best Practices

- Scaling Up
  - How many jobs will be checkpointing?
  - How big are the checkpoint files?
  - How much data is that total?

Avoid:

- Filling up the SPOOL directory.
- Transferring large checkpoint files.

- Checkpoint Frequency
  - How long does it take to produce a checkpoint and resume?
  - How likely is your job to be interrupted?

Avoid:

- Spending more time checkpointing than running.
- Jobs that will never reach a checkpoint.

Alternative Checkpointing Method

- If code can't exit after each checkpoint, but only run + checkpoint continuously, transfer of checkpoint files can be triggered by eviction.
- Search for "when_to_transfer_output" on the condor_submit manual page; read about ON_EXIT_OR EVICT
- This method of backing up checkpoint files is less resilient, as it won't work for other job interruption reasons (hardware issues, killed processes, held jobs)

Resources

- HTCondor Manual
  - Manual > Users' Manual > Self Checkpointing Applications
  - https://htcondor.readthedocs.io/en/latest/users-manual/self-checkpointing-applications.html
- Materials from the OSG Virtual School 2021
  - OSG Virtual School > Materials > Overview or Checkpointing Exercises
  - https://opensciencegrid.org/virtual-school-2021/materials/#self-checkpointing-for-long-running-jobs

Acknowledgements

Todd L Miller; Christina Koch

This work is supported by NSF under Grant Nos. 2030508, 1836650, and 1148698.

Questions?