

# Getting Started with Flux for EECS 470

*Prepared for [Dr. Mark Brehob](#) by [Kyle Smith](#), Winter 2014*

# Table of Contents

## Introduction

### 1. Overview

### 2. Getting Access

- 2.1 Two Factor Authentication (2fa)
- 2.2 Synopsys Licensing
- 2.3 Flux Allocation

### 3. Running Jobs (synthesis)

- 3.1 Flux Login Servers
- 3.2 Writing Job Scripts
- 3.3 PBS and useful tools

### 4. TL;DR

### 5. Practical Examples (and l33t h4x)

- 5.1 Manually Submit (\_Using Better Build System\_)
- 5.2 Psuedo-daemon monitoring `git diff`

### 6. Appendices

- 6.1 Node Configuration

# Getting Started with Flux for EECS 470

*Prepared for [Dr. Mark Brehob](#) by [Kyle Smith](#), Winter 2014*

Many, *many* thanks to [Dan Barker](#), [Bennet Faube](#), and the mmmkk team. This effort wouldn't have been possible without your time, patience, and support.

This entire guide and all related code is available on GitHub in the following repositories:

- <https://github.com/EECS470WN14-group3/flux-documentation>
- <https://github.com/EECS470WN14-group3/flux-practical-scripts>

Please let us know if you find any errors, or have any ideas about how to improve this guide for future students! Pull requests are welcome and encouraged.

# Overview

# Overview

This short guide is intended to get students, GSIs, and professors, started with synthesis builds using [Synopsys Design Compiler](#) on [Flux](#).

If you haven't used [Design Compiler](#) yet ( `make syn` with the [Better Build System](#)), I'd recommend attending lab or reading through the GSI's [Synthesis and Makefile overview](#) before proceeding.

Before you get too deep in this guide, it's worth noting that the CAEN [High Performance Computing \(HPC\) Group](#) does a fairly reasonable job of documenting Flux on their [homepage](#). If you want to figure everything out the hard way, jump on over to [Getting Started at the CAC](#). This guide **does** cover a lot of the same content; however, in order to get you up and running **quickly**, I place a **heavy emphasis on EECS 470 specifics** and leave out a lot of general working knowledge.

In case you were handed this guide and have no idea why, a bit of context: the HPC Group describes Flux as:

The U-M's campus Linux-based high performance computing cluster

At the time of writing (Winter 2014), Flux is comprised:

- 632 standard multi-core compute nodes (8,016 total cores)
  - 4GB of RAM per core
- 10 large-memory compute nodes (32 or 40 cores per node)
  - 1TB of total RAM per system
- 640TB of high-speed scratch storage

Full stats and configuration options are available on the HPC website: <http://caen.engin.umich.edu/hpc/flux-configuration>.

## TL;DR

Flux is giant cluster of computers with lots of parallel computing power. This guide will teach you how to run synthesis on Flux.

---

### Useful Links

- HPC Homepage: <http://caen.engin.umich.edu/hpc/overview>
- Flux Specs and Configurations <http://caen.engin.umich.edu/hpc/flux-configuration>

# Getting Access

# Getting Access

Flux, along with the other [HPC Systems](#), is wrapped up in a metric ton of red tape, which is to say, **getting access is a time consuming process**. This chapter will walk through the 4 primary steps required to gain access to: authenticate with, and launch synthesis on, the Flux login servers. They are:

1. Obtaining an MToken and activating Two Factor Authentication
2. Requesting access to Synopsys
3. Requesting an HPC user account
4. Creating a Flux Allocation

## Useful Contacts

Two HPC members were pivotal to the success of our pilot test during the Winter 2014 term: [Dan Barker](#) and [Bennet Faube](#).

Dan was our first point of contact and took care of setting up our allocation and adding authorized users. He was extremely responsive, thorough, and supportive.

- Dan Barker
- [danbarke@umich.edu](mailto:danbarke@umich.edu)
- (734) 763-9840

Bennet was our lifeline from HPC and the single reason we were ever able to complete a synthesis build. Bennet has extensive knowledge of the entire platform and took care of all the nitty gritty versioning and licensing issues we faced with Synopsys. **He's the man.**

- Bennet Fauber
- [bennet@umich.edu](mailto:bennet@umich.edu)
- (734) 764-6226

---

### Useful Links

- Getting Started at the CAC: <http://cac.engin.umich.edu/getting-started>
- Managing a Flux Project: <http://arc.research.umich.edu/flux-and-other-hpc-resources/flux/managing-a-flux-project/>

# Two Factor Authentication (2fa)

Flux is a [batch processing system](#). More about what that means later. For now, just understand that, in order to use Flux, you'll need access to two things:

- The [HPC Login Nodes](#) which are simply SSH login servers (like [login.engin.umich.edu](#) ).
- A [Flux allocation](#) to run jobs on.

In order to access [HPC Login Nodes](#) (covered in more detail in the next chapter), you must first obtain an [MToken](#). **Skip ahead** if you already have an activated MToken.

## 1. What's an MToken?

MTokens are Michigan-branded [RSA SecurIDs](#), which are the most popular hardware [security tokens](#) in mass circulation. For a quick overview of 2fa, check out [CAEN's intro video on YouTube](#) Additionally, the [ITS MToken website](#) lists which university resources require 2fa for access.

The below instructions were lifted from the following CAEN and CAC pages:

- <http://www.itcs.umich.edu/itcsdocs/s4394/>
- <http://cac.engin.umich.edu/resources/login-nodes/tfa>

## 2. Obtain an MToken

Go to the [Computer Showcase Repair Center](#) in **Pierpont Commons**. If you're not on north campus, find another Distribution Center from [the list](#)

## 3. Activate Your MToken

With your shiny new Mtoken **in hand**, open <http://mtoken.umich.edu/> and follow the activation instructions

### ***If you can't visit in person***

**Use this as a last resort.** Send an email to [4help@umich.edu](mailto:4help@umich.edu) and ask real nice. With some luck, they'll physically mail you the dongle or send a software token via email.

---

### Useful Links

- MToken activation / management site: <http://mtoken.umich.edu>
- ITS 2fs guide: <http://www.itcs.umich.edu/itcsdocs/s4394/>
- CAC 2fa guide: <http://cac.engin.umich.edu/resources/login-nodes/tfa>



# Synopsys Licensing and Access

In order to use Synopsys on Flux (and off-campus), you must submit an agreement to the [Synopsys off-campus license terms](#).

At the time of writing, this was done using [this Google Doc](#):

- <https://docs.google.com/a/umich.edu/forms/d/1AGb6w2LbMf5wOI84SRNPEaNOXn76PgeMIp5Qiq1dTzI/viewform>

If for any reason that link stops working, try the [Access to Synopsys Tools](#) website, or visit the [CAEN contact page](#).

---

## Useful Links

- <http://caen.engin.umich.edu/software/access-to-synopsys-tools>
- <https://docs.google.com/a/umich.edu/forms/d/1AGb6w2LbMf5wOI84SRNPEaNOXn76PgeMIp5Qiq1dTzI/viewform>

# Flux Allocation and User Account

To submit jobs on Flux, you'll need an activated HPC user account **and** an allocation that lists that account as an authorized user.

## 1. Request an Account

Everyone needs an HPC user account. Request one by [completing this form](#)

## 2. For Students

Kindly ask your professor to email [hpc-support@umich.edu](mailto:hpc-support@umich.edu) to add your account to the access list

## 3. For Professors

Request a new allocation by emailing [hpc-support@umich.edu](mailto:hpc-support@umich.edu) the following:

- number of cores needed
- start date and duration needed
- list of users that should have access to submit jobs
- list of users that should be admins and able to change user list and allocation properties
- shortcode for funding (current pricing on the [hardware services site](#))

**NOTE:** Most allocation requests require one business day for setup.

## Winter 2014 Pricing Specifics

The following information was provided by [Dan Barker](#) on April 15, 2014:

- Flux allocations can only be purchased in month long increments.
- The cost of a Flux allocation to a member of the College of Engineering is \$6.60 per core per month.
- [We] usually have the cores ready by the next business day after we receive a request.
- Standard Flux (\$11.72/core/month @ 4 GB/core)
- Big Memory Flux (\$23.82/core/month @ 25 GB/core)
- LSA, CoE, and the Medical School will cost-share 44%, reducing the cost to their researchers to
  - \$6.60/core/month for Standard Flux
  - \$13.30/core/month for Big Memory Flux
- The allocation [could] take 1-2 business days to set up, so if you specify a start date of today, you may "lose" a day's worth of access.

---

### Useful Links

- Terminology & How Flux Works: <http://arc.research.umich.edu/flux-and-other-hpc-resources/flux/how-flux-works/>
- Allocation information: <http://arc.research.umich.edu/flux-and-other-hpc-resources/flux/managing-a-flux-project/>
- Current prices: <http://arc.research.umich.edu/flux-and-other-hpc-resources/flux/hardware-services/>
- Allocation Usage Report: <https://mreports.umich.edu/mreports/pages/Flux.aspx>

## Running Jobs (synthesis)

# Running Jobs (*synthesis*)

Flux processes jobs using a batch system. If you've never used a batch system before, it's useful to think of jobs as backgrounded processes (eg: `./my_script &`); batch systems, unlike the interactive shell that you're likely accommodated to (eg: `BASH`), queue and execute unattended programs without requiring (or, generally speaking, allowing) user interaction.

## Topics Covered

This chapter is the meat-and-potatoes of this guide and will cover the following topics:

- Logging into Flux
- Writing scripts to run jobs
- Submitting jobs, `PBS`, and other useful tools

## The Concept

Using the login server, you'll issue a command that will submit a job to the worker queue. Once the queue selects your job for execution, a worker node will start whatever program you've given the job. While the worker node is executing your program you'll have (generally speaking) very limited knowledge of what is actually happening (eg: what state the program is in, what messages are being printed to `STDOUT` or `STDERR`).

## The Implementation

In the context of EECS 470, this means running a synthesis build will involve logging into Flux, issuing a command to submit `make syn` as the program to run, and waiting until the job has completed. If all goes well, you can start synthesis on Flux, log off, grab a bite to eat, log back in an hour or two later, and view the `proc.rep` file to make sure everything work as expected. **No more all-nighters in 1695 waiting for builds to complete!**

---

### Useful Links

- Batch Processing: [http://en.wikipedia.org/wiki/Batch\\_processing](http://en.wikipedia.org/wiki/Batch_processing)
- Job Schedulers: [http://en.wikipedia.org/wiki/Job\\_scheduler](http://en.wikipedia.org/wiki/Job_scheduler)

# Flux Login Servers

At this point, you should be ready to log into Flux! Congratulations! Before proceeding, please **make sure you have the following**:

- Obtained and activated an MToken
- Completed the Synopsys off-campus license terms agreement
- Registered for an HPC User account
- Received confirmation details for your Flux allocation, including:
  - Project name (eg: `some-username_flux` )
  - Authorized users list (just make sure you're on it!)

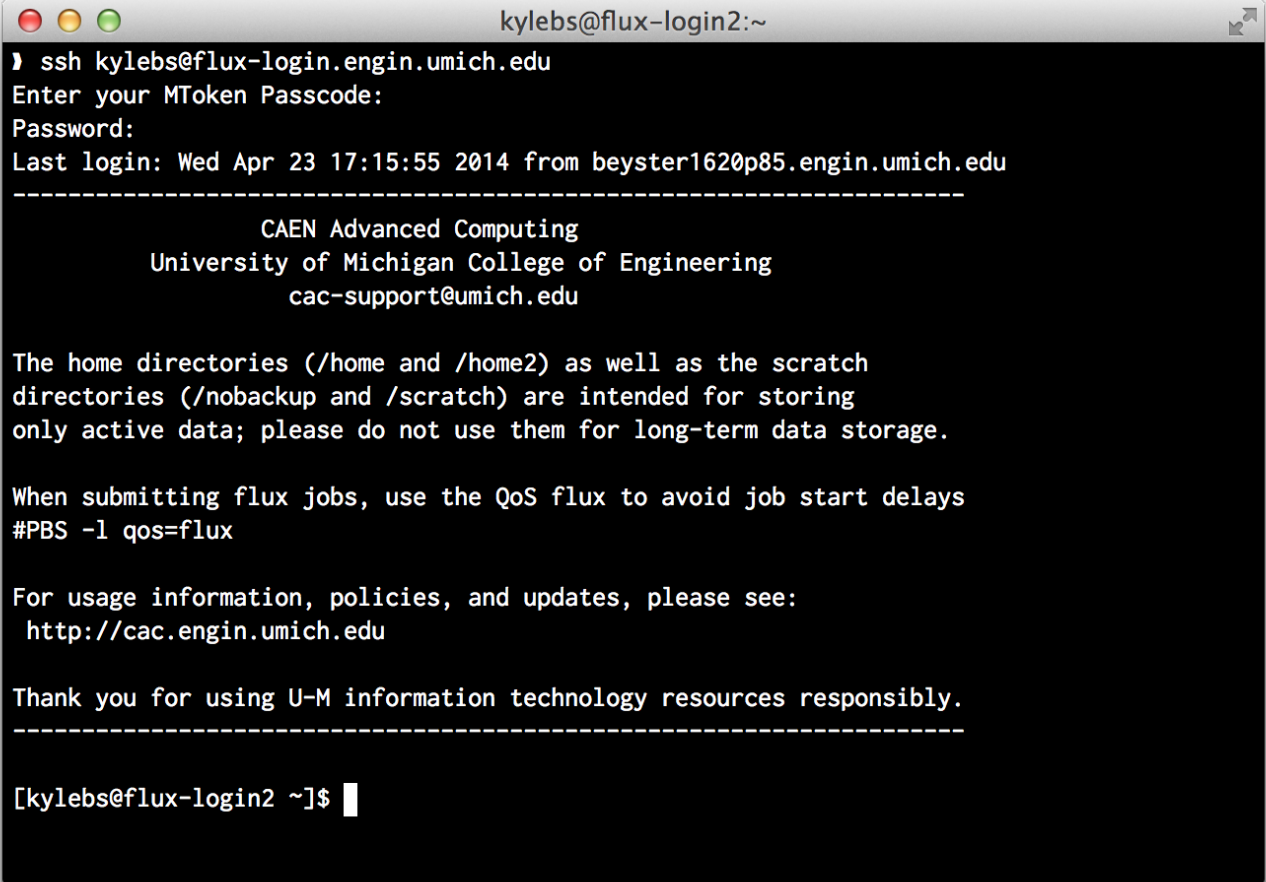
If you are missing **any** of the above, please go back and review the previous chapters. You will need **all** of these things to submit a job!

## SSH Into a Flux Login Server

The [login nodes](#) for Flux are accessible via hostname at `flux-login.engin.umich.edu` . Make sure you're **on campus** and issue the following command:

```
$ ssh your-username@flux-login.engin.umich.edu
```

If all goes well, you will be prompted for an MToken code, followed by your password. If you see the following welcome banner (or something like it), give yourself a high five!



```
kylebs@flux-login2:~  
» ssh kylebs@flux-login.engin.umich.edu  
Enter your MToken Passcode:  
Password:  
Last login: Wed Apr 23 17:15:55 2014 from beyster1620p85.engin.umich.edu  
-----  
                CAEN Advanced Computing  
      University of Michigan College of Engineering  
                cac-support@umich.edu  
  
The home directories (/home and /home2) as well as the scratch  
directories (/nobackup and /scratch) are intended for storing  
only active data; please do not use them for long-term data storage.  
  
When submitting flux jobs, use the QoS flux to avoid job start delays  
#PBS -l qos=flux  
  
For usage information, policies, and updates, please see:  
http://cac.engin.umich.edu  
  
Thank you for using U-M information technology resources responsibly.  
-----  
[kylebs@flux-login2 ~]$
```

# Login Server Guidelines

Remember that the [login nodes](#) are intended for just that: logging in and submitting jobs to the cluster. You should **never run resource-intensive commands at the login shell**; that's what the worker nodes are for!

If you need to run an interactive script that **is** resource intensive (can't think of any reason you'd do this in EECS 470), [submit an interactive job](#).

## Protips

If, like me, typing out `flux-login.engin.umich.edu` over and over annoys you, open `~/.ssh/config` with your favorite editor and add the following lines:

```
Host flux
  Hostname flux-login.engin.umich.edu
  User your-username
```

Now when you want to log into Flux, you need only type:

```
$ ssh flux
```

If you'd like to try setting up [SSH multiplexing](#), CAC has a mini-guide on the [two factor authentication page](#)

---

## Useful Links

- <http://cac.engin.umich.edu/resources/login-nodes>

# Writing Job Scripts

Now that you can access the login nodes, it's time to start writing job scripts. Job scripts are what you submit to the job scheduler to be put in the worker queue. On Flux, we use [Portable Batch System \(PBS\)](#) (covered in more detail next chapter). For EECS 470, our job scripts will be simple shell scripts that **copy files** and **invoke `make`**.

## Quick Disclaimer

**This isn't supposed to be a guide on shell scripting.** If you already have a good handle on command-line tools like `make`, understanding job scripts should be cake; however, if you don't feel comfortable with the Black Screen of Wonder™, you should copy-paste the example code provided later on, **OR** find a friend and work through it. **I won't spend time explaining standard commands or scripting practices.** If you want to strengthen your BASH-fu, you might consider spending some time reverse-engineering posts on [commandlinefu.com](http://commandlinefu.com).

## Hello World

No guide would be complete without a "Hello World" example, so here's ours. Consider the below shell script, titled `hello-world-job.sh`:

```
#!/bin/bash
#PBS -A brehob_flux
#PBS -l qos=flux
#PBS -q flux

echo "Hello World!"
```

As you can see, short of those 3 weird `#PBS` comment lines, this is just your regular old, run-of-the-mill bash script. We'll cover **how** to submit this job in the next section. For now, **assume that running this job produces 2 new text files** in the working directory: `hello-world-job.sh.o12253000` and `hello-world-job.sh.e12253000`. These files contain output from `STDOUT` and `STDERR`, respectively. More on these files in just a minute.

## Job Attributes

Let's turn your attention back to those 3 weird lines that start with `#PBS`. These are obviously comments of some sort (the line **does** starts with a `#`), but what are they and why do we need them for our simple "Hello World" job? **These comments are special directives** to the job scheduler that specify [job attributes](#). There are many attributes that you can set and a full list is available on the [CAC PBS reference page](#).

While many attributes are optional, **all jobs must** specify Queue, Account, and Quality of Service (QOS). Queue and QOS should stick to the default values listed below while Account should use the allocation name you were given via email.

- Queue `-q flux`
- Account `-A {your allocation name here}`
- Quality of Service (QOS) `-l qos=flux`

## Log Files

You seem like a smart person; I'm guessing you've figured out that it's no coincidence they're named `{job-script}.{ 'o' or 'e' }{numbers}`. The default behavior of PBS is to dump two files into the directory that the job was **submitted from**. The `{numbers}` portion of the filename is the **Job ID** which is established by PBS for tracking purposes and returned to the user upon successful submission. The `{ 'o' or 'e' }` portion indicates that these files contain the "output" and "error" logs, respectively. Remember how we said PBS runs "unattended programs?" Since we can't (generally speaking) watch `STDOUT` and `STDERR` while a job is running, PBS pipes everything from `STDOUT` into the `{ 'o' }` log and everything from `STDERR` into the `{ 'e' }` log. This way, we can review what the program printed to the terminal while we weren't watching.

## Environment Modules

Flux is an incredibly complex system and must support the needs of many users. Due to the nature of High Performance Computing, the HPC Group has to be extremely careful with software versioning; many jobs will run for weeks, months, and even years.

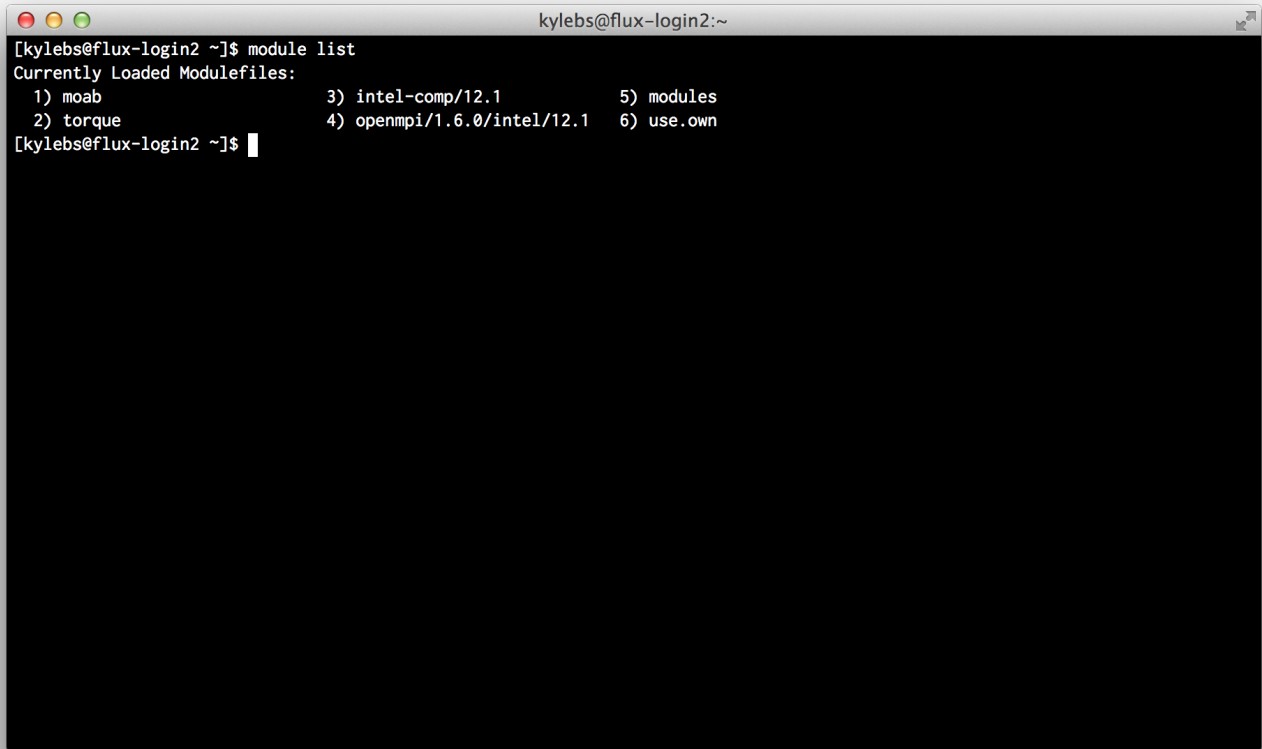
In order to better-accommodate the wide range of users and software requirements (versions), Flux makes use of [environment modules](#) to **dynamically load and unload common software packages**.

As a user of Flux, you need to be aware of this because, as you'll soon find, just because a software package is **installed** (available), does **NOT** mean it's accessible for use (loaded). **Synopsis is an example of one package that is available but NOT loaded by default.**

More command line tools will be covered in the next section; however, since you need to understand **module** to work through the next example job, let's quickly describe it here. We'll be looking at the 4 most-common commands: **list**, **available**, **load**, and **unload**.

1. **list** : List packages that are currently loaded into the environment:

```
$ module list
```



A terminal window titled 'kylebs@flux-login2:~' showing the output of the 'module list' command. The output lists currently loaded modulefiles in a numbered list format:

```
[kylebs@flux-login2 ~]$ module list
Currently Loaded Modulefiles:
  1) moab                  3) intel-comp/12.1       5) modules
  2) torque                4) openmpi/1.6.0/intel/12.1 6) use.own
[kylebs@flux-login2 ~]$
```

2. **available** : List packages that are installed but **not** currently loaded:

```
$ module available
```



```
kylebs@flux-login2:~$ module available

----- /home/kylebs/privatemodules -----
null

----- /usr/flux/software/rhel6/Modules/modulefiles -----
abaqus/6.11          flexlm/11.7.0.0(default)  mathematica/7.0          nastran/2012(default)
abaqus/6.12(default) fluent/14.0(default)     mathematica/8.0(default) pgi/10.5
abaqus/6.12-3       fluent/14.5             mathematica/9.0          pgi/11.8
ampl/12.2(default)  fluent/15.0             mathematica/9.0.1       pgi/12.10
ampl/20130903       gaussian/09-revc(default) matlab/2010b            pgi/12.3(default)
ampl/20131112       hfss/13.0               matlab/2012a(default)   pgi/12.9
ansys/14.0(default) hfss/14.0(default)      matlab/2012b            pgi/13.5
ansys/14.5          hfss/15.0               matlab/2013a            sas/9.3(default)
ansys/15.0          hfss/2014.0.0           maxwell/15.0.1(default) starccm/7.02(default)
cfx/14.0(default)   idl/8.2(default)        mkl/10.3.7(default)    starccm/7.04
cfx/14.5           imsl/fnl/7.0.0/intel/12.1 mkl/11.0               starccm/7.06
cfx/15.0           intel-comp/11.1         mkl/11.1               starccm/8.04
chemkin/10112       intel-comp/12.1(default) msmodel/5.5(default)   stata/12(default)
comsol/3.5a         intel-comp/13.0         msmodel/6.0             stata/13
comsol/4.3(default) intel-comp/13.0.1       msmodel/6.1             stata-mp/12
cplex/12.4(default) intel-comp/13.1         nag/5.1                 stata-mp/13
cubit/13.2(default) lsdyna/971-R5.1.1       nag/5.2(default)        synopsys/2009.06-SP2
ddt/4.0(default)    lsdyna/971-R610(default) naglib/23/clib/gcc       synopsys/2012.06
dytran/2012(default) maple/16(default)       naglib/23/clib/icc      synopsys/2013.03-SP1
epd/7.3-2           maple/17                naglib/23/flib/gfortran  tecplot/2012(default)
epd/7.3-2a(default) marc/2012(default)      naglib/23/flib/ifort    vcs/2013.06-sp1

----- /home/software/rhel6/Modules/modulefiles -----
R/2.15.0           hdf5/1.8.9/pgi/12.3-par  openmpi/1.6.1/gcc/4.4.6
```

3. **Load** : Load an **available** package into the current environment. For example, to load the default version of synopsys and vcs:

```
$ module load synopsys vcs
```

```
kylebs@flux-login2:~  
[kylebs@flux-login2 ~]$ module load synopsys vcs  
  
NOTE: All Synopsys tools have special access restrictions. Please see  
  
http://caen.engin.umich.edu/software/access-to-synopsys-tools  
  
for information about access restrictions. You will need to obtain  
the "Off-campus network access from home" per those instructions,  
and once you have logged into the host oncampus-course.engin.umich.edu  
send e-mail to flux-support@umich.edu and request that you be added to  
the synopsys group.  
  
[kylebs@flux-login2 ~]$ module list  
Currently Loaded Modulefiles:  
  1) moab                3) intel-comp/12.1      5) modules              7) synopsys/2013.03-SP1  
  2) torque              4) openmpi/1.6.0/intel/12.1  6) use.own              8) vcs/2013.06-sp1  
[kylebs@flux-login2 ~]$
```

As you can see, `2013.03-SP1` and `2013.06-sp1` are the default versions of Synopsys and VCS, respectively.

4. **unload** : Remove a loaded package from the current environment. For example, to unload the current versions of synopsys and vcs:

```
$ module unload synopsys vcs
```

And to specifically load in older versions by tag:

```
$ module load synopsys/2012.06 vcs/2011.03
```

```
kylebs@flux-login2:~  
[kylebs@flux-login2 ~]$ module list  
Currently Loaded Modulefiles:  
  1) moab                      3) intel-comp/12.1           5) modules                   7) synopsys/2013.03-SP1  
  2) torque                   4) openmpi/1.6.0/intel/12.1  6) use.own                   8) vcs/2013.06-sp1  
[kylebs@flux-login2 ~]$ module unload synopsys vcs  
[kylebs@flux-login2 ~]$ module load synopsys/2012.06 vcs/2011.03  
  
NOTE: All Synopsys tools have special access restrictions. Please see  
  
http://caen.engin.umich.edu/software/access-to-synopsys-tools  
  
for information about access restrictions. You will need to obtain  
the "Off-campus network access from home" per those instructions,  
and once you have logged into the host oncampus-course.engin.umich.edu  
send e-mail to flux-support@umich.edu and request that you be added to  
the synopsys group.  
  
[kylebs@flux-login2 ~]$ module list  
Currently Loaded Modulefiles:  
  1) moab                      3) intel-comp/12.1           5) modules                   7) synopsys/2012.06  
  2) torque                   4) openmpi/1.6.0/intel/12.1  6) use.own                   8) vcs/2011.03  
[kylebs@flux-login2 ~]$
```

As you can see, we unload `2013.03-SP1` and `2013.06-sp1` and load in legacy versions `2012.06` and `2011.03`.

## Job Restrictions & Limitations

At this point, it's probably a good idea to describe some of the restrictions and limitations on jobs. As you tweak and develop job scripts for **your** projects, keep the following in mind.

- **No internet access inside jobs.** While you can access the interwebs from the login nodes, jobs are sandboxed. For the record, I can't find any reference that confirms this fact, but I certainly witnessed and experienced it.  
*If* you simply need to grab something from the interwebs before the job runs (eg: clone a repository, grab the latest copy of some external file), you can wrap the actual job script with a job "submission" script that will perform these tasks **before** the job enters the queue.
- **No automatic AFS access.** Meaning, by default, the Flux login nodes do **not** generate AFS `tokens` for you. If you need to access AFS space, you'll need to obtain the appropriate AFS tokens manually:

```
# list current tokens  
$ tokens  
  
# if "--End of list--" is displayed with no tokens, then you need to generate a token  
$ aklog -c umich.edu  
  
# now, unless something went wrong, you should have a token  
$ tokens
```

Fair warning that this **may** require you to enter your password, which means you should definitely keep this out of job scripts. Entering passwords into scripts == sketchy.

If you need something from AFS space for a job, obtain a token at the login shell and wrap the job script a job "submission" script (same as #1 above).

- Accessing AFS and **network space is super slow** and could easily become a bottleneck in your batch jobs. Fortunately, HPC provides us with some "scratch storage" space. 640TB of high-speed scratch storage space to be exact. Therefore, it's a good idea (and best practice) to copy any files needed by your job to your scratch space **before** the job executes (eg:

using a job "submission" script again).

Every authorized user on a project is granted scratch space in `/scratch/{account}/{user}` . For example, my scratch space in Dr. Brehob's trial allocation is located in `/scratch/brehob_flux/kylebs` .

## Simple 2-Script Synthesis Job

Now that we've covered the fundamentals, let's consider a basic synthesis job that uses 2 different scripts (a very common structure):

- 1 for pre-submission file operations
- 1 for the actual job

**NOTE: If run as-is, this job will fail! Keep reading to find out why**

Here's the pre-submission script, `pre-sub.sh` :

```
#!/bin/bash

export JOB_SCRIPT=job-script.sh
export PROJ_DIR=`pwd`
export SCRATCH_SPACE="/scratch/brehob_flux/$USER"

# Need to load synopsys and vcs modules
module load synopsys/2013.03-SP1
module load vcs/2013.06-sp1

echo "Submitting job..."
JOB_ID=`qsub $JOB_SCRIPT`
if [ "$?" -ne '0' ]; then
    echo "Error: Could not submit job via qsub"
    exit 1
fi

unset JOB_SCRIPT
unset PROJ_DIR
unset SCRATCH_SPACE

echo "JobID: $JOB_ID"
```

Here's the actual job script, `job-script.sh` :

```
#!/bin/bash
#PBS -S /bin/sh
#PBS -N eecs470synth
#PBS -A brehob_flux
#PBS -q flux
#PBS -l qos=flux,nodes=1:ppn=12,mem=47gb,pmem=4000mb,walltime=05:00:00
#PBS -V

# copy the project dir to the scratch space
temp_dir="$SCRATCH_SPACE/$PBS_JOBID"
mkdir -p "$temp_dir"
cd "$temp_dir"
cp -R "$PROJ_DIR" .
make syn
```

In a nutshell, to launch a new job, you'd move both scripts into the root of your project directory and start the `pre-sub.sh` by running the following command from the login shell:

```
$ ./pre-sub.sh
```

A few things would then happen:

1. `$PROJ_DIR` would be set to the current working directory (the one where you Makefile lives)
2. `$SCRATCH_SPACE` would be set to the absolute path to your scratch space. In this example, my allocation is named `brehob_flux`
3. Synopsys and VCS are loaded into the environment (specific versions are targeted)
4. A mysterious new command, `qsub`, is issued and the job is submitted. If the return code ( `$?` ) is anything besides `0`, then the job submission failed and we print an error message and exit.

Assuming the job was submitted and queued up successfully, then, sometime later, a worker node would start `job-script.sh` and a few more things would happen:

1. A few **new job parameters are used**, notably:
  - `-V`, which tells the scheduler to propagate the current environment (the one that submitted the job to begin with) into this one. This is **extremely important** because it will make the exported environment variables (ie: `$PROJ_DIR` and `$SCRATCH_SPACE`) available to the job, **and** will ensure that previously loaded modules (ie: Synopsys and VCS) are also available.
  - `-l nodes=1:ppn=12,mem47gb,pmem=4000mb,walltime=05:00:00`, which sets up the node architecture. [Bennet Fauber](#) did an excellent job at describing how these values affect the job performance; you can read an excerpt from [his email in the appendix](#). Unless you've been gifted with a [high memory](#) allocation, stick to these defaults; they're tried and true.
2. A new directory is created in your scratch space with a name equal to the job's ID.
3. The project directory that the job was submitted from is copied to scratch space.
4. Synthesis is started by invoking `make syn`.

Now for the bad news: unfortunately, while this looks good and seems correct, there are hidden dependencies in the synthesis files for Better Build System that will cause synthesis to fail. Specifically, `scripts/default.tcl` contains the following line of code:

```
set search_path "/afs/umich.edu/class/eecs470/lib/synopsys/"
```

Essentially, **EECS 470 uses custom libraries** that must be loaded in at synthesis time. Since the files are **on AFS**, the **load will fail** because, as we know, there isn't a valid AFS token in the job environment.

Fortunately, there's an easy (but dirty) solution: **copy all the included files** to the scratch space **before** the job executes, and use `sed` or another tool to replace all instances of the old path with the new one. For now, just be aware of this issue; we'll give some example code to take care of this in the final chapter, "Practical Examples".

---

## Useful Links

- <http://cac.engin.umich.edu/resources/software/pbs>
- <http://cac.engin.umich.edu/resources/systems/nyx/pbs>

# PBS and Useful Tools

Working with batch jobs can be challenging. It's important to have a solid understanding of the tools available, and, even more importantly, to know where to find more information when necessary.

There's a [great list of common commands](#) provided by CAC available online at:

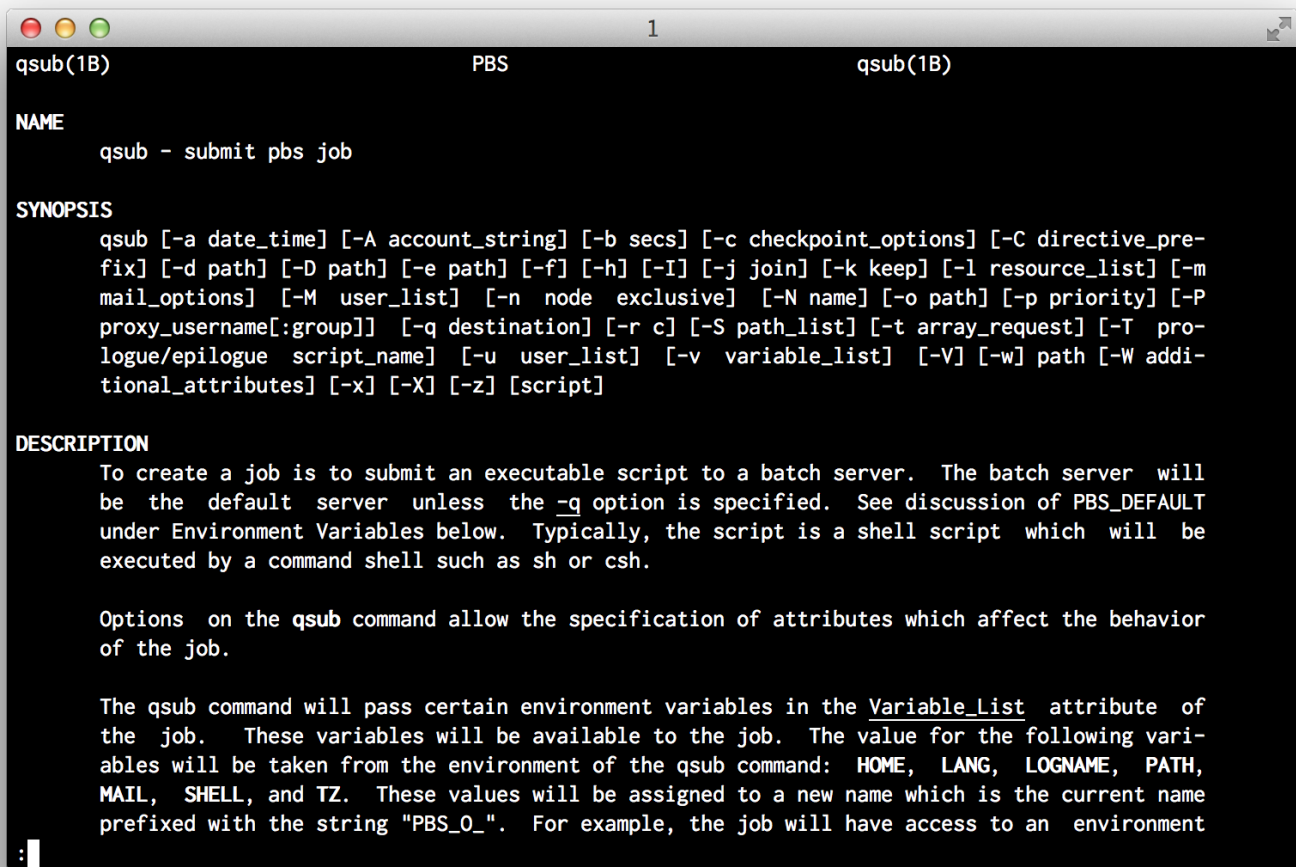
- <https://docs.google.com/a/umich.edu/viewer?v=v&pid=sites&srcid=dW1pY2guZWRIfGVuZ2luLWNhY3xneDpjYmRlMGQyMzVIY2FhYjg>.

This section attempts to quickly cover the basics of PBS to get you up and running ASAP.

## 1. Submitting jobs with `qsub`

`qsub` allows you to submit new jobs. A super brief example of this was shown in the previous section.

A quick view of the man page ( `man qsub` ) reveals a frightening number of flag modifiers:



```
qsub(1B)                                PBS                                qsub(1B)

NAME
    qsub - submit pbs job

SYNOPSIS
    qsub [-a date_time] [-A account_string] [-b secs] [-c checkpoint_options] [-C directive_pre-
    fix] [-d path] [-D path] [-e path] [-f] [-h] [-I] [-j join] [-k keep] [-l resource_list] [-m
    mail_options] [-M user_list] [-n node exclusive] [-N name] [-o path] [-p priority] [-P
    proxy_username[:group]] [-q destination] [-r c] [-S path_list] [-t array_request] [-T pro-
    logue/epilogue script_name] [-u user_list] [-v variable_list] [-V] [-w] path [-W addi-
    tional_attributes] [-x] [-X] [-z] [script]

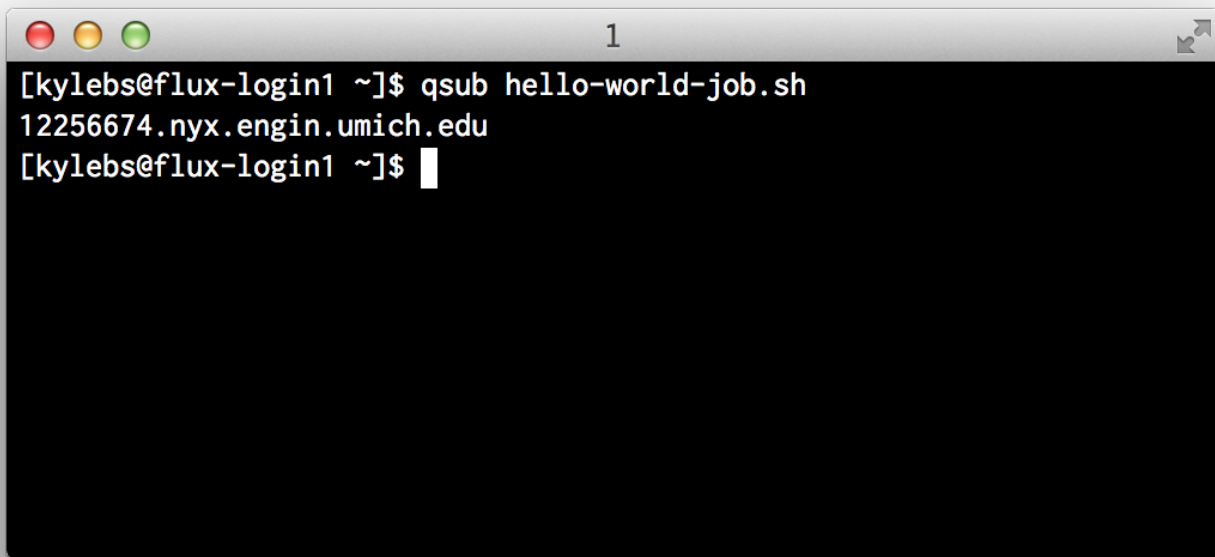
DESCRIPTION
    To create a job is to submit an executable script to a batch server. The batch server will
    be the default server unless the -q option is specified. See discussion of PBS_DEFAULT
    under Environment Variables below. Typically, the script is a shell script which will be
    executed by a command shell such as sh or csh.

    Options on the qsub command allow the specification of attributes which affect the behavior
    of the job.

    The qsub command will pass certain environment variables in the Variable_List attribute of
    the job. These variables will be available to the job. The value for the following vari-
    ables will be taken from the environment of the qsub command: HOME, LANG, LOGNAME, PATH,
    MAIL, SHELL, and TZ. These values will be assigned to a new name which is the current name
    prefixed with the string "PBS_O_". For example, the job will have access to an environment
    :
```

While **most** flags are optional, **all** jobs **must** specify Queue ( `-q` ), Account ( `-A` ), and Quality of Service (QOS) ( `-l qos=[name]` ). Hopefully this seems familiar and you've realized that those strange comments in our `hello-world-job.sh` script were simply a **different** way to pass these required job attributes to the scheduler. We could submit the original "Hello World" script (shown below) using `qsub hello-world-job.sh`, or we could remove all those comment directives and pass them into the `qsub` command directly.

On a successful call to `qsub`, the job ID is returned to `STDOUT`:

A terminal window with a title bar containing three colored circles (red, yellow, green) and the number '1'. The terminal text shows a user at a prompt submitting a job with the command 'qsub hello-world-job.sh'. The output shows the job ID '12256674.nyx.engin.umich.edu' and returns to the prompt.

```
[kylebs@flux-login1 ~]$ qsub hello-world-job.sh
12256674.nyx.engin.umich.edu
[kylebs@flux-login1 ~]$
```

Using in-file comment directives

```
#!/bin/bash
##### The original hello-world-job.sh script
#PBS -A brehob_flux
#PBS -l qos=flux
#PBS -q flux

echo "Hello World!"
```

Submit `hello-world-job.sh` using: `qsub hello-world-job.sh` . OR use the modified version below:

```
$ qsub hello-world-job.sh
```

Using command-line flags

```
#!/bin/bash
##### hello-world-job-nocomments.sh with the comment directives removed
echo "Hello World!"
```

And submit `hello-world-job-nocomments.sh` using:

```
qsub -A brehob_flux -l qos=flux -q flux hello-world-job-nocomments.sh
```

```
$ qsub -A brehob_flux -l qos=flux -q flux hello-world-job-nocomments.sh
```

Which way is best?

There's no good answer. If you have a job that will always be submitted with the exact same attributes and doesn't need to be flexible, why not hardcode them to prevent dumb mistakes? If you want to be able to invoke the job script from a "helper" script (like our `pre-sub.sh` script from earlier), then you'll need to use command line flags, as the comment directives won't allow for variable expansion (because they're **comments**).

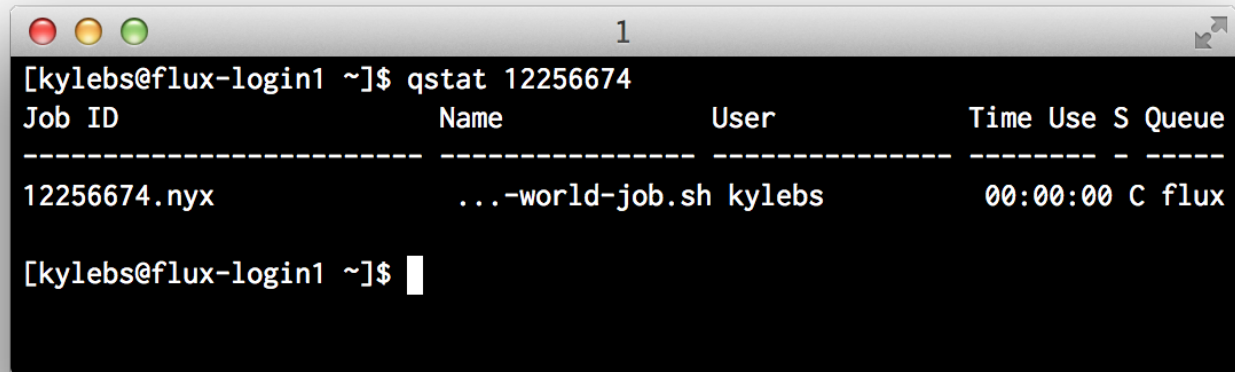
## 2. Checking jobs with `qstat`

`qstat` allows you to check the status of submitted jobs. Running with no args is practically useless as it will dump every job on every queue from every user. There are two cases where you'll commonly use `qstat` :

- Check the status of a specific job
- View all of your jobs

#### Checking a Specific Job (By Job ID)

```
# check a specific job with ID 12256674
$ qstat 12256674
```



```
[kylebs@flux-login1 ~]$ qstat 12256674
Job ID              Name              User              Time Use S Queue
-----
12256674.nyx        ...-world-job.sh kylebs            00:00:00 C flux

[kylebs@flux-login1 ~]$
```

As you can see, qstat gives us back some basic information about the requested job:

- Job ID
- Job Name (specified with `-N [name]` )
- User that submitted the job
- Time Use (for currently-running jobs, the amount of time the job has been running)
- S, or Status (Primary states are: `C` for completed, `Q` for queued, and `R` for running. See the man page for a list of all possible states)
- Queue describes which worker queue the job was submitted to

#### Checking All Jobs For a User

```
# check all jobs for a specific user, you!
$ qstat -u $USER
```



```
[kylebs@flux-login1 ~]$ qstat -u $USER

nyx.engin.umich.edu:

Job ID                Username   Queue    Jobname          SessID  NDS   TSK   Req'd   Req'd   S    Elap
-----
12256673.nyx.engin.umi kylebs    flux     hello-world-job. 107053   --    --    768mb   01:00:00 C    --
12256674.nyx.engin.umi kylebs    flux     hello-world-job. 107103   --    --    768mb   01:00:00 C    --
12256677.nyx.engin.umi kylebs    flux     hello-world-job. 92387    --    --    768mb   01:00:00 C    --
12256678.nyx.engin.umi kylebs    flux     hello-world-job. 27259    --    --    768mb   01:00:00 C    --
12256682.nyx.engin.umi kylebs    flux     hello-world-job. 0         --    --    768mb   00:01:00 R   00:00:16
12256683.nyx.engin.umi kylebs    flux     hello-world-job. --        --    --    768mb   00:01:00 Q    --
[kylebs@flux-login1 ~]$
```

Very similar to the specific job status shown above but with some re-wording and the addition of:

- SessID, or SessionID (I have no idea what this is for)
- NDS, or Number of Nodes (specify with `-l nodes=X` )
- TSK, or Number of Tasks (CPUs) (specify with `-l procs=X` or `-l nodes=X:ppn=Y` where `ppn` stands for "Processors Per Node")

### 3. Introspect running jobs with `qpeek`

Often times, it's useful to be able to check on a long-running process to make sure everything is executing as expected. A healthy mix of `echo` statements in your job scripts and the occasional `qpeek` can be indispensable when it comes down to identifying bugs in your jobs.

In the example below, I "accidentally" submit a job with an endless loop and peek in on it to see what's going on.

```
$ qpeek 12256691
```

```
1
Still running at time: 1398381074...
Still running at time: 1398381075...
Still running at time: 1398381076...
Still running at time: 1398381077...
Still running at time: 1398381078...
Still running at time: 1398381079...
Still running at time: 1398381080...
Still running at time: 1398381081...
Still running at time: 1398381082...
Still running at time: 1398381083...
Still running at time: 1398381084...
Still running at time: 1398381085...
Still running at time: 1398381086...
Still running at time: 1398381087...
Still running at time: 1398381088...
Still running at time: 1398381089...
Still running at time: 1398381090...
Still running at time: 1398381091...
Still running at time: 1398381092...
Still running at time: 1398381093...
Still running at time: 1398381094...
Still running at time: 1398381095...
Still running at time: 1398381096...
Still running at time: 1398381097...
[kylebs@flux-login1 ~]$
```

#### 4. Cancelling jobs with `qdel`

Sometimes you need to cancel a job after it has been submitted. It **generally** doesn't matter what state the job is currently in, so long as you are the owner, you should be able to cancel it with `qdel`.

In the example above with `qpeek`, we saw that `looper` was mis-behaving. We can use `qdel` to kill that job, make the necessary changes, and resubmit:

```
# kill job 12256691
$ qdel 12256691
```

```
[kylebs@flux-login1 ~]$ qdel 12256691
[kylebs@flux-login1 ~]$ qstat 12256691
Job ID          Name          User          Time Use S Queue
-----
12256691.nyx    eecs470doc    kylebs        00:00:00 C flux
[kylebs@flux-login1 ~]$
```

## Handy One-liners

Here are a few handy bash one-liners I like to use when working with PBS:

```
# leave a terminal open with this running to constantly monitor my jobs
$ while [ 1 ]; do qstat -u $USER; sleep 1; done

# list all jobs in a CSV-ish format that's easy to parse
$ qstat -u $USER | grep -E '^^[0-9]+' | tr -s '[:blank:]' ','

# list all jobs that aren't complete
$ qstat -u $USER | grep -vE '^00 C' | grep -E '^^[0-9]+'

# find all of my queued jobs and kill them
$ qstat -u $USER | grep -E '^^[0-9]+' | tr -s '[:blank:]' ',' | cut -d',' -f1,10 | grep -E 'Q$' | cut -d'.' -f1 |
xargs -I {} qdel {}

# find all of my running jobs and kill them
$ qstat -u $USER | grep -E '^^[0-9]+' | tr -s '[:blank:]' ',' | cut -d',' -f1,10 | grep -E 'R$' | cut -d'.' -f1 |
xargs -I {} qdel {}
```

---

### Useful Links

- CAC PBS Guide for nyx: <http://cac.engin.umich.edu/resources/systems/nyx/pbs>
- CAC General PBS Guide: <http://cac.engin.umich.edu/resources/software/pbs>

- **Free** HPC Video Tutorials: <http://arc.research.umich.edu/training-workshops/online-training-resources/>
- CAC \*nix Quick Reference Sheet: <https://docs.google.com/a/umich.edu/viewer?a=v&pid=sites&srcid=dW1pY2guZWV1fGVuZ2luLWNhY3xneDpjYmRlMGQyMzVlY2FhYjg>

TL;DR

# TL;DR This Entire Guide in 1 Epic List

## Get Access

- Obtain an [MToken](#) from the [Computer Showcase Repair Center](#) in **Pierpont Commons**. All pickup locations on [the list](#).
- Activate your MToken at <http://mtoken.umich.edu/>. At the time of writing, this was done using [this Google Doc](#):
- Accept the [Synopsis off-campus license terms](#) using this form:
  - <https://docs.google.com/a/umich.edu/forms/d/1AGb6w2LbMf5wOI84SRNPEaNOXn76PgeMIp5Qiq1dTzI/viewform>
- Request an HPC user account with this form:
  - <https://www.engin.umich.edu/form/cacaccountapplication>
- Request a new Flux allocation (professors) or access to an existing one (students)
  - **Students:** have your professor email [hpc-support@umich.edu](mailto:hpc-support@umich.edu)
  - **Professors:** request a new allocation by emailing [hpc-support@umich.edu](mailto:hpc-support@umich.edu) the following:
    - number of cores needed
    - start date and duration needed
    - list of users (**students**) that should have access to submit jobs
    - list of users that should be admins (**you and GSIs**) and able to change user list and allocation properties
    - shortcode for funding
- **Wait one full business day (at least)**

## Running Jobs

- If you don't know the shell, **stop and go learn it**.
- From **on campus**, SSH into the [Flux login nodes](#) using `flux-login.engin.umich.edu`.
- Submit a new job using `$ qsub -A prof-username_flux -l qos=flux -q flux your-job-script.sh`
- See the CAC \*nix Quick Reference Sheet for more commands:
  - <https://docs.google.com/a/umich.edu/viewer?a=v&pid=sites&srcid=dW1pY2guZWZlLWNhY3xneDpjYmRlMGQyMzVIY2FhYjg>
- Download the example Flux job scripts and drop them into your `scripts` directory in Better Build:
  - <https://github.com/EECS470WN14-group3/flux-practical-scripts>

## Helpful People

- Dan Barker for **general questions and getting an allocation set up**.
  - [danbarke@umich.edu](mailto:danbarke@umich.edu)
  - (734) 763-9840
- Bennet Fauber for **Flux-specifics and issues with Synopsis**.
  - [bennet@umich.edu](mailto:bennet@umich.edu)
  - (734) 764-6226
- Kyle Smith's voicemail for **all the questions you have because you skipped the rest of the guide and only read this page**
- Kyle Smith for **all legitimate questions / concerns / beers**.
  - [@knksmith57](https://twitter.com/knksmith57)
  - (248) 491-3202

# Practical Examples (and l33t h4x)

# Manually Submit (*Using Better Build System*)

This is a [forked](#) version of [Joshua Smith](#)'s original [synth.sh](#) and [pbs.sh](#) setup. It has been updated to utilize the new-er [scratch space](#) that didn't exist (or he wasn't using?) in 2010.

## scripts/synth.sh

```
#!/bin/bash -l
# Created Fall 2010 by: Joshua Smith <smjoshua@umich.edu>
# Updated Winter 2014 by: Kyle Smith <kylebs@umich.edu>

SCRIPTS_DIR="$(pwd -P)"
pushd ../

PROJ_DIR="$(pwd -P)"
EECS470_LIBS_DIR="/afs/umich.edu/class/eecs470/lib"

#####
## PBS Script Parameters                                     ##
## Full reference available at:                             ##
## http://cac.engin.umich.edu/resources/software/pbs         ##
#####
PBS_SHELL=/bin/sh
PBS_JOB_NAME="470-synth"
PBS_ACCOUNT="brehob_flux"
PBS_JOB_ATTRS="qos=flux,nodes=1:ppn=12,mem=47gb,pmem=4000mb,walltime=10:00:00"
PBS_QUEUE="flux"
PBS_EMAIL_ADDR="kylebs@umich.edu"
PBS_EMAIL_OPTS="abe"
PBS_FLAGS="-V"
PBS_FILE="{SCRIPTS_DIR}/pbs.sh"

#####
## Copy EECS 470 Synopsys Libs Locally From: ${EECS470_LIBS_DIR} ##
#####
# Scratch space ftw
LOCAL_EECS470_DIR="{HOME}/eecs470-lib"
SCRATCH_DIR="/scratch/{PBS_ACCOUNT}/{USER}"
if ! [ -d "${LOCAL_EECS470_DIR}" ]; then
    echo "Warning: Local EECS 470 Synopsys library directory doesn't exist, attempting to copy..."
    cp -R "${EECS470_LIBS_DIR}" "${LOCAL_EECS470_DIR}"

    if [ "$?" -ne '0' ]; then
        echo "Error: Failed to copy EECS 470 Synopsys library to local directory:"
        echo "    ${EECS470_LIBS_DIR} -> ${LOCAL_EECS470_DIR}"
        popd
        exit 1
    fi
fi

# Export some environment variables so PBS can access them when job runs
export EECS470_LIBS_DIR
export LOCAL_EECS470_DIR
export PROJ_DIR
export PBS_ACCOUNT
export SCRATCH_DIR

# Need to load these so paths of dc_shell/vcs are found
module load synopsys/2013.03-SP1
module load vcs/2013.06-sp1
```



```

# Submit the batch job
echo "Submitting batch job..."
JOB_ID=`qsub -S $PBS_SHELL -N $PBS_JOB_NAME -A $PBS_ACCOUNT -l $PBS_JOB_ATTRS -q $PBS_QUEUE -M $PBS_EMAIL_ADDR -m
$PBS_EMAIL_OPTS $PBS_FLAGS $PBS_FILE`
if [ "$?" -ne '0' ]; then
    echo "Error: Could not submit job via qsub"
    popd
    exit 1
fi
echo "Submitted batch job, id=$JOB_ID"

# clean up stuff
unset EECS470_LIBS_DIR
unset LOCAL_EECS470_DIR
unset PROJ_DIR
unset PBS_ACCOUNT
unset SCRATCH_DIR

popd

# to make parsing of result easier...
echo "jobID=$JOB_ID"

```

## scripts/pbs.sh

```

#!/bin/bash
# Created Fall 2010 by: Joshua Smith <smjoshua@umich.edu>
# Updated Winter 2014 by: Kyle Smith <kylebs@umich.edu>
#
# This script is executed to run the synthesis job on the computing node.

pushd ../

# Create a local directory for this run and copy project files into it
TMP_DIR=${SCRATCH_DIR}/${PBS_JOBID}
echo "PBS - Copying project files to scratch space: ${TMP_DIR}..."
mkdir -p $TMP_DIR
cd $TMP_DIR
rsync -avz $PROJ_DIR/ .
if [ "$?" -ne '0' ]; then
    echo "PBS - Error while trying to copy project files to tmp"
    cd
    rm -rf $TMP_DIR
    popd
    exit 1
fi

# Copy EECS470 libs for Synopsys
LIBS_DIR=${SCRATCH_DIR}/eecs470-libs
echo "PBS - Copying eeecs470 Synopsys libs to scratch space..."
mkdir -p $LIBS_DIR
cd $LIBS_DIR

# this assumes you have a copy of the EECS 470 Synopsys libs in ${LOCAL_EECS470_DIR}!
rsync -avz ${LOCAL_EECS470_DIR} .

if [ "$?" -ne '0' ]; then
    echo "PBS - Error while trying to copy eeecs470 Synopsys libs to scratch space using source directory: ${LOCAL_EECS470_DIR}"
    echo "PBS --- Have you copied the EECS 470 libs to your HPC home directory?"
    echo "PBS --- Try running:"

```

```

echo "PBS --- $ cd"
echo "PBS --- $ aklog -c umich.edu"
echo "PBS --- $ cp -R /afs/umich.edu/class/eecs470/lib eeecs470-libs"
cd
rm -rf $TMP_DIR
popd
exit 1
fi

# Find & Replace old paths with the new local one
echo "PBS - Re-linking all eeecs470 Synopsys libs to the updated local path @ ${LIBS_DIR}..."
cd $TMP_DIR
find . -type f -exec sed -i "s,${EECS470_LIBS_DIR},${LIBS_DIR},g" {} \;

# Run synthesis
echo "PBS - Running synthesis..."
make syn

# Clean up simulation output
make clean

# Copy `proc.rep` result file back to home space
cd $TMP_DIR
cp syn/proc.rep ${PROJ_DIR}/proc.${PBS_JOBID}.rep
if [ "$?" -ne '0' ]; then
    echo "PBS - Error while trying to copy proc.rep back to project home"
    cd
    popd
    exit 1
fi

echo "PBS - Done!"

popd

```

## Pseudo-daemon Monitoring with `git diff`

This is an enhancement to the code in `manual-better-build.md` that adds intelligent `git diff` checking and a "pseudo-daemon" that runs in the background and checks for new commits before auto-launching a new build.

There's also a section commented out that hits a remote REST server with job status updates so that we can track status without having to hop on a terminal. If you want implement the REST server for remote status tracking, hit me up and I can get you the code base --> [@knksmith57](#).

### `scripts/daemon.sh`

```

#!/bin/bash

PBS_SCRIPT="synth.sh"
JOB_LOG_FILE="jobs-for-commits.log"

export REMOTE_NAME="origin"
export SYNTH_BRANCH="flux"

PENDING_UPDATE=0
PENDING_UPDATE_JOBID=0

while [ 1 ]; do
    # get a comma-separated list of all jobs that aren't complete

```

```

jobs="$(qstat -u $USER | grep -vE '00 C' | grep -E '^[0-9]+' | tr -s '[:blank:]' ',')"

# check remote git repository for a commit change
update_detected=0
local_commit="$(git rev-list --max-count=1 $REMOTE_NAME/$SYNTH_BRANCH)"
remote_commit="$(git ls-remote --heads $REMOTE_NAME $SYNTH_BRANCH)"
remote_issue="$?"
remote_commit="$(echo "$remote_commit" | cut -f1)"
[ "$remote_issue" -eq 0 ] && [ "$local_commit" != "$remote_commit" ] && update_detected=1;

# send updates back to remote REST server
# curl --data-urlencode "v=1" \
#      --data-urlencode "jobs=${jobs}" \
#      --data-urlencode "log=$(cat $JOB_LOG_FILE)" \
#      --data-urlencode "update_requested=${update_requested}" \
#      http://api.your-server.com/ &> /dev/null

# if update detected, queue up job!
if [ $update_detected -ne 0 ]; then
    echo "update detected!"

    # write the update activity to the local log
    echo "update detected @ $(date +%s)" >> updates.log;

    PENDING_UPDATE="$remote_commit"
    PENDING_UPDATE_JOBID=0

    # kill any queued jobs-- this one is obviously newer
    for job in $jobs; do
        # get the status of this job (Q == Queued, C == Complete, R == Running)
        status="$(echo "$job" | cut -d',' -f10)"
        jobid="$(echo "$job" | cut -d',' -f1 | cut -d'.' -f1)"

        if [ "$status" == "Q" ]; then
            echo "detected previously queued job: $jobid, killing it...";
            qdel $jobid;
        fi
    done

    # update the repo, pull down the latest
    git fetch $REMOTE_NAME
    # git reset --hard
    # git checkout $SYNTH_BRANCH
    # git reset --hard
    # git pull $REMOTE_NAME $SYNTH_BRANCH

    # grab the new commit hash to report back with
    export COMMIT_HASH="$(git rev-parse --short "$remote_commit")"

    # submit the new job using l33t h@ck5
    # NOTE: this git logic is pretty stupid and hacked together. don't just
    # blindly put this into production without testing + tweaking first.
    git checkout "${SYNTH_BRANCH}_base" $PBS_SCRIPT
    submit_result="$( ./PBS_SCRIPT )"
    if [ "$?" -ne '0' ]; then
        echo "Error: Failed to submit job with PBS script: $PBS_SCRIPT";
    else
        # parse out the job id from the output of the PBS script
        jobid="$( echo "$submit_result" | tail -1 | cut -d'=' -f2 | cut -d'.' -f1)"
        submit_time=$(date +%s);

        # log the jobid:commit_hash relationship
        echo "$jobid,$COMMIT_HASH,$submit_time" >> $JOB_LOG_FILE;
    fi
fi

```

```

    fi
else
    sleep 1;
fi

git reset --hard "${SYNTH_BRANCH}_base"
done

```

## scripts/synth.sh

```

#!/bin/bash -l
# Created Fall 2010 by: Joshua Smith <smjoshua@umich.edu>
# Updated Winter 2014 by: Kyle Smith <kylebs@umich.edu>

SCRIPTS_DIR="$(pwd -P)"
pushd ../

PROJ_DIR="$(pwd -P)"
EECS470_LIBS_DIR="/afs/umich.edu/class/eecs470/lib"

#####
## PBS Script Parameters                                     ##
## Full reference available at:                             ##
## http://cac.engin.umich.edu/resources/software/pbs         ##
#####
PBS_SHELL=/bin/sh
PBS_JOB_NAME="470-synth"
PBS_ACCOUNT="brehob_flux"
PBS_JOB_ATTRS="qos=flux,nodes=1:ppn=12,mem=47gb,pmem=4000mb,walltime=10:00:00"
PBS_QUEUE="flux"
PBS_EMAIL_ADDR="kylebs@umich.edu"
PBS_EMAIL_OPTS="abe"
PBS_FLAGS="-V"
PBS_FILE="${SCRIPTS_DIR}/pbs.sh"

#####
## Copy EECS 470 Synopsys Libs Locally From: ${EECS470_LIBS_DIR} ##
#####
# Scratch space ftw
LOCAL_EECS470_DIR="${HOME}/eecs470-lib"
SCRATCH_DIR="/scratch/${PBS_ACCOUNT}/${USER}"
if ! [ -d "${LOCAL_EECS470_DIR}" ]; then
    echo "Warning: Local EECS 470 Synopsys library directory doesn't exist, attempting to copy..."
    cp -R "${EECS470_LIBS_DIR}" "${LOCAL_EECS470_DIR}"

    if [ "$?" -ne '0' ]; then
        echo "Error: Failed to copy EECS 470 Synopsys library to local directory:"
        echo "    ${EECS470_LIBS_DIR} -> ${LOCAL_EECS470_DIR}"
        popd
        exit 1
    fi
fi

# Export some environment variables so PBS can access them when job runs
export EECS470_LIBS_DIR
export LOCAL_EECS470_DIR
export PROJ_DIR
export PBS_ACCOUNT
export SCRATCH_DIR

# Need to load these so paths of dc_shell/vcs are found

```

```

module load synopsys/2013.03-SP1
module load vcs/2013.06-sp1

# Submit the batch job
echo "Submitting batch job..."
JOB_ID=`qsub -S $PBS_SHELL -N $PBS_JOB_NAME -A $PBS_ACCOUNT -l $PBS_JOB_ATTRS -q $PBS_QUEUE -M $PBS_EMAIL_ADDR -m
$PBS_EMAIL_OPTS $PBS_FLAGS $PBS_FILE`
if [ "$?" -ne '0' ]; then
    echo "Error: Could not submit job via qsub"
    popd
    exit 1
fi
echo "Submitted batch job, id=$JOB_ID"

# clean up stuff
unset EECS470_LIBS_DIR
unset LOCAL_EECS470_DIR
unset PROJ_DIR
unset PBS_ACCOUNT
unset SCRATCH_DIR

popd

# to make parsing of result easier...
echo "jobID=$JOB_ID"

```

## scripts/pbs.sh

```

#!/bin/bash
# Created Fall 2010 by: Joshua Smith <smjoshua@umich.edu>
# Updated Winter 2014 by: Kyle Smith <kylebs@umich.edu>
#
# This script is executed to run the synthesis job on the computing node.

pushd ../

# Create a local directory for this run and copy project files into it
TMP_DIR=${SCRATCH_DIR}/${PBS_JOBID}
echo "PBS - Copying project files to scratch space: ${TMP_DIR}..."
mkdir -p $TMP_DIR
cd $TMP_DIR
rsync -avz $PROJ_DIR/ .
if [ "$?" -ne '0' ]; then
    echo "PBS - Error while trying to copy project files to tmp"
    cd
    rm -rf $TMP_DIR
    popd
    exit 1
fi

# Copy EECS470 libs for Synopsys
LIBS_DIR=${SCRATCH_DIR}/eecs470-libs
echo "PBS - Copying eeecs470 Synopsys libs to scratch space..."
mkdir -p $LIBS_DIR
cd $LIBS_DIR

# this assumes you have a copy of the EECS 470 Synopsys libs in ${LOCAL_EECS470_DIR}!
rsync -avz ${LOCAL_EECS470_DIR} .

if [ "$?" -ne '0' ]; then
    echo "PBS - Error while trying to copy eeecs470 Synopsys libs to scratch space using source directory: ${LOCAL_E
ECS470_DIR}"

```

```
echo "PBS --- Have you copied the EECS 470 libs to your HPC home directory?"
echo "PBS --- Try running:"
echo "PBS --- $ cd"
echo "PBS --- $ aklog -c umich.edu"
echo "PBS --- $ cp -R /afs/umich.edu/class/eecs470/lib eeecs470-libs"
cd
rm -rf $TMP_DIR
popd
exit 1
fi

# Find & Replace old paths with the new local one
echo "PBS - Re-linking all eeecs470 Synopsys libs to the updated local path @ ${LIBS_DIR}..."
cd $TMP_DIR
find . -type f -exec sed -i "s,${EECS470_LIBS_DIR},${LIBS_DIR},g" {} \;

# Run synthesis
echo "PBS - Running synthesis..."
make syn

# Clean up simulation output
make clean

# Copy `proc.rep` result file back to home space
cd $TMP_DIR
cp syn/proc.rep ${PROJ_DIR}/proc.${PBS_JOBID}.rep
if [ "$?" -ne '0' ]; then
    echo "PBS - Error while trying to copy proc.rep back to project home"
    cd
    popd
    exit 1
fi

echo "PBS - Done!"

popd
```

# Appendices

Kyle,

I ramble on in the first answer. Perservere, and the question about interactive is answered way down there.

As I'm sure you know, there are two memory models in parallel computing: the memory is shared among the processes, which all just use different address ranges, or it is distributed, which is just copying the chunk you need to where you're going to use it. Basically, that boils down to 'same machine' or 'many machines'. Same machine is faster, almost always.

When you ask for nodes=1:ppn=12, you're asking for one machine with at least 12 processors per machine. All of the machines we have in the public pool have at least 12 processors, so any of them will be able to run your job.

When you ask for nodes=1:ppn=16, you're asking for one machine with at least 16 processors per machine. About 1/4 of our machines are configured that way, and people know those are faster processors, so more people ask for them by name. You're waiting for all the processors on one whole machine to come free, but the scheduler is juggling many thousands of jobs that don't need the whole machine, so at least one processor on each machine is already committed for a long time into the future.

When you request procs=16, without any node requirements, then you're almost certainly now in the distributed memory situation, so the overhead of copying data among the machines (up to 16) will have an effect -- at its worst, it will take longer to copy data around than it will to calculate the results. How much time it will add depends on the nature of the problem, how well can it be divided, and the efficiency of the algorithms used to partition and copy things around.

Using procs=N is the fastest because we have so many jobs running, and finding some combination across a bunch of machines is pretty quick and easy. That's ideal for jobs where each process is, effectively, a job of its own and doesn't have to communicate with the other processes. Many programs have a lot more chatter among the processes while the program is running, so they all wait on the slowest running processor at any given step (which may be a different processor on a different node for any given step).

You'll probably find that nodes=1:ppn=12 will be the best combination for most things, given what you've said already. You don't have an unlimited number of processors, so you can't simply increase the number of processors to compensate for the slow-down of using processors on different machines. You also don't have unlimited time, so it's probably better to get your work done than to figure out how to do it faster when you get to it.

Memory can be requested by either mem=N or pmem=M, which is total memory across the whole job or minimum memory guaranteed to each processor within a job. If you ask for only one node, then all the memory in the job is pooled on the one node, so they can be roughly equivalent, i.e.,  $N = M * \text{procs}$ . It's more complicated when you are using more than one node. If you ask for mem=16gb along with cores=16, then the scheduler **could** give you one processor of the bunch with 500 MB, one with 1.5 GB, and the rest with 1 GB. If you're using one node consistently, you could use either, but remember that our 12 core nodes have 48 GB of memory, some of which is needed by the operating system and cluster management software, so you shouldn't ask for nodes=1:ppn=12,mem=48gb, as that will be able to run only on machines with more than 12 cores, so you're back to waiting in line for a 12 of 16 cores on a popular machine type. Better to ask for either nodes=1:ppn=12,mem=47gb or nodes=1:ppn=12,pmem=4000mb, which will leave sufficient room for the OS overhead.