Batch Scheduling: A Fresh Approach

Nicholas P. Cardo, Sterling Software, Inc., Numerical Aerodynamic Simulation Facility, NASA Ames Research Center, Moffett Field, CA

ABSTRACT: *The Network Queueing System (NQS) was designed to schedule jobs based on limits within queues. As systems obtain more memory, the number of queues increased to take advantage of the added memory resource. The problem now becomes too many queues. Having a large number of queues provides users with the capability to gain an unfair advantage over other users by tailoring their job to fit in an empty queue. Additionally, the large number of queues becomes confusing to the user community. The High Speed Processors (HSP) group at the Numerical Aerodynamics Simulation (NAS) Facility at NASA Ames Research Center developed a new approach to batch job scheduling. This new method reduces the number of queues required by eliminating the need for queues based on resource limits. The scheduler examines each request for the necessary resources before initiating the job. Also additional user limits at the complex level were added to provide a fairness to all users. Additional tools which include user job reordering are under development to work with the new scheduler. This paper discusses the objectives, design and implementation results of this new scheduler.*

With increasing memory and cpu capacities, alternative batch job scheduling mechanisms are needed. The method of taking first come first served and not examining current system utilization has become outdated. The time has come to investigate improvements in batch scheduling.

1 Purpose

A new batch scheduler has been designed to address short comings of existing scheduling options and implement enhancements that benefit the user community.

Better handling of large and multitasking jobs provide for improved throughput. The ability to handle these jobs has been built into the scheduler and reduces any manual intervention required.

Another change is the scheduling of jobs based on submit limits rather than by queue limits. This provides the ability to schedule jobs better and reduce turnaround time. Improved system utilization can also be gained by scheduling based on job requirements.

The ability to submit large numbers of jobs has also been added. This will allow a user to load the system for overnight, weekend or holiday work. Not only does this add flexibility to the user but also allows the system to contain enough jobs to keep from going idle.

The implementation would also reduce the number of queues required to sustain the user load. A minimal queue configuration is necessary to reduce the chances of queue gaming and general confusion.

2 Objectives/Requirements

The objectives/requirements of the new batch scheduler include:

1. Minimal NQS modifications.
2. Minimal queue configuration.
3. Easily supported.
4. Customizable.
5. Usable on all HSP systems.
6. 0% Idle
7. Minimal Swapping
8. Regular Scheduling of Large Jobs
9. Good Throughput for Debug Jobs
10. Regular Running of MT Jobs
11. High Priority Jobs
12. Equality between Small and Large User Groups
13. Additional Utilities

2.1 Minimal NQS modifications

By minimizing the number of local modifications to NQS, there would be less of a chance of introducing conflicting code into NQS. It is possible for a local modification to conflict with a Cray released fix. Also, modifications to NQS are complicated and time consuming due to the complexity and size of NQS.

2.1.1 User Exits

User Exits were introduced at UNICOS 8.0. The premace behind them is to provide entry points into key parts of programs. This provides the capability of adding functionality to programs without having to modify source code. Another benefit to using these is ease of sharing code with other sites, including binary only sites.

The batch scheduler design first attempted to utilize User Exits. The library `/usr/lib/libuex.a` contains all the user exits. When NQS is built, `libuex.a` is included as one of the libraries. User exits are provided at the following locations within NQS:

- NQS daemon packet. Allows sites to add functionality when a packet arrives.
- NQS destination ordering. Allows sites to control the order of pipe queue destinations within NQS.
- NQS job selection. Allows sites to determine if the request chosen by NQS should be started. The order in which requests are started by NQS can be customized.
- NQS job initiation. Allows a user exit before job initiation.
- NQS job termination. Allows a user exit after job termination.
- NQS `qmgr` command. Allows additional functionality when a `qmgr` command is going to be processed.
- qsub directives. Allows user exits before the first `#QSUB` directive, on `#QSUB` directive, and after the last `#QSUB` directive.
- NQS startup. Allows the user to perform processing during the NQS startup.
- NQS shutdown. Allows the user to perform processing during the NQS shutdown.
- Job submission.

A major effort was required to port existing modifications in NQS to a new release. This increases the length of time before the new release can be released into production. By utilizing user exits, the scheduling portion of NQS can be modified without actually modifying NQS. This would reduce the effort necessary to port mods to new releases of NQS and UNICOS.

The investigation into the utilization of User Exits revealed weaknesses that were not able to be overcome. In particuliar, they seemed to only provide a mechanism for augmenting operations on existing parameters and functionality. The introduction of new command line arguements and the altering of key internal functions were not possible. This approach was abandoned.

2.1.2 Modifications

Modifications would now be required to add new functionality to NQS. Although most modifications were minor, there did exist a more complicated portion. The addition of user complex limits needed to be added to NQS in a way that would make upgrades easy. When possible, new functions being added to NQS were placed at the end of source modules. Rather than insert additional code into existing functions, the insertion of a function call was made. This simplified upgrade efforts tremendously. The port from UNICOS 8.0.2 to UNICOS 8.0.3 took under 2 hours.

2.2 Minimal Queue Configuration

The objective is to provide a queue configuration with a minimal number of queues. This simplifies job submission, maintenance and adds equality to jobs, regardless of job size.

Figure 1 shows the previous queue configuration.

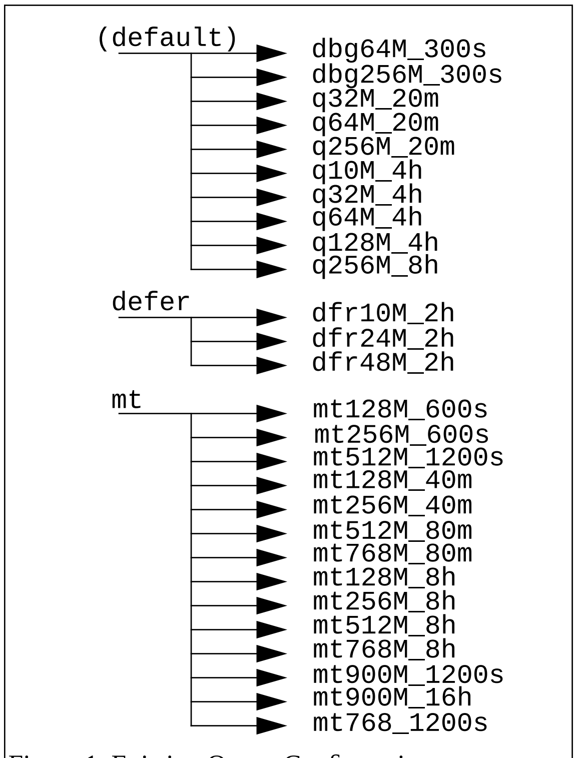

Figure 1, Existing Queue Configuration

As can be seen, there are 27 regular production queues. This is in addition to system maintenance and special access queues.

Under the new scheduler, the queues can be greatly simplified. Figure 2 shows the new queue configuration after implementation of the new batch scheduler. Previously the queues were arranged by memory and cpu limits. One queue will be used for each of multitasking, large memory multitasking, deferred, debug, and regular batch jobs. Scheduling will no longer be based on queue limits but rather the actual requested limits of the job. Previously, a 130MW request would have

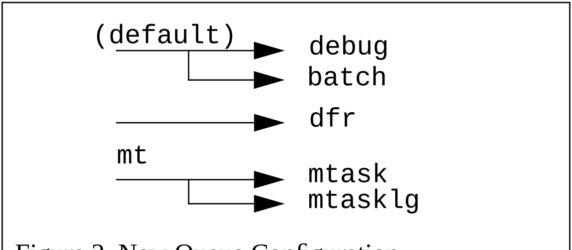

Figure 2, New Queue Configuration

been queued in the 256MW queue. Under the new batch scheduler, the 130MW request will not be scheduled as 256MW. By scheduling based on requested limits, memory can be more efficiently utilized, a better job mix can be obtained, and a greater throughput achieved.

2.3 *Easily Supported*

The scheduling algorithm is able to initiate jobs with the goal being to keep the system as busy as possible without causing a degradation in service. It is also able to run multitasking and large memory jobs throughout the day. The main objective is that this scheduling system should be self sufficient and require as little manual intervention as possible.

Although the previous goals were very similar, the differences are with how multitasking and large jobs are handled. Under the previous setup, multitasking jobs typically required manual intervention and usually only ran during non-prime time hours. The difference is to attempt to schedule large and multitasking jobs around the clock.

2.4 *Customizable*

Although there exists a goal to use the same batch scheduling system on both C90 systems, it is recognized that there are differences in the machines. The new batch scheduler allows parameter changes to customize the scheduling algorithm to work best on each system.

These customizable parameters deal with specific policies of the respective systems as well as hardware configuration differences. Customizable parameters include memory, job size, run limits, and system utilization rates. Some parameters would be configurable through `qmgr`, the NQS queue manager.

An example of a memory difference is, what is considered to be a large job on one C90, would be considered a small job on the other. This is due to physical hardware differences between the two systems. In the current configuration, one C90 has 128MW while the other has 1024MW of memory. The job selection algorithm is configurable to work with either system.

Additionally, scheduling parameters are easily changed. A configuration file is maintained and can be altered at any time. The scheduler will reread the configuration file when it receives a SIGHUP signal.

2.5 *Usable on all HSP's*

In order to minimize the impact on support staff, the same scheduler is utilized on both C90 systems with configuration changes to fit that system. This reduces the need to maintain separate modifications on both systems for NQS.

2.6 *0% Idle*

By not keeping enough jobs running, or by having the wrong job mix, idle time can occur. Ideally, by scheduling the job mix, and by keeping an optimum number of jobs running, idle time should be less than 4% and optimally 0%. The scheduler initiates jobs so that the system does not end up with a poor mix of jobs (ie., too many large jobs). In addition, no noticable degradation in interactive performance has been observed.

2.7 *Minimal Swapping*

By running too many large memory jobs, swapping will increase and system overhead will be incurred. This in turn causes idle time on the system.

Swapping is controlled in three ways.

1. There is limit how much total memory batch jobs are allowed to consume.
2. The size of a large job and the limit on how many can run simultaneously are configurable.
3. The swap-in rate is constantly monitored. A configurable parameter exists to set what is considered an acceptable swap rate.

2.8 *Regular Scheduling of Large Jobs*

By carefully controlling the job mix, large jobs can run at regular intervals. This will improve overall turnaround time for large memory jobs. A minimum of one large memory job will always be attempted to keep running. A maximum number of large memory jobs to run is configurable.

2.9 *Good Throughput for Debug Jobs*

For the most part, debug jobs are small and fast. The debug queue will operate as an unscheduled queue. That is, the queue will have small run and memory limits. Additionally there is a limit for the total memory used by all jobs running in the debug queue. This will provide constant flow for debug jobs while keeping them from severely impacting the system.

2.10 *Regular Running of Multitasking Jobs*

Multitasking is encouraged to achieve high performance in solving computational problems. To accommodate this, a mechanism for running multitasking jobs throughout the day is being incorporated into the scheduler. During prime time hours, these jobs may be restricted to a limited number of cpu's and memory. Large mulitasking jobs are only run during non-prime time hours.

A parameter has been added to the `qsub` command which allows the user to specify the number of cpu's required. This would allow scheduling of the job effectively since the full requirements are known. When a process is evaluated for initiation, the number of cpu's required is taken into consideration.

By knowing the number of cpu's required for the job before the job starts, it becomes possible to predict contention for cpu's. For example, knowing that a job will require 16 processors would be used as an indicator when to run the job. A job requiring only 2 cpu's can be scheduled much easier than a 16 cpu job. By only connecting to 2 cpu's, there is minimal contention for the cpu. Whereas if 16 cpu's were required, there will be cpu contention if the job is run at the wrong time.

2.11 Processing of High Priority Jobs

There will always be a need for high priority jobs. Currently the use of special queues for high priority work is done. Although the new scheduler will not eliminate the requirements for special queues, a way to flag a job as high priority is being investigated. This would allow high priority jobs to run in the same queues as other jobs. By running special jobs in the same batch queue, job scheduling can be performed on the special jobs. Currently, special queues are enabled and jobs start as they are submitted. By marking the job as next to run it may provide for a better performing system.

2.12 Equality Between Small and Large User Groups.

In an effort to insure equality between large and small user groups, the new scheduler utilizes a three state scheduling process. Batch jobs will flow through three states - the *pending* state where all jobs enter, the *queued* state where only jobs eligible to be selected for running exist, and the *running* state for jobs in execution.

Under normal conditions, the following rules apply.

- Each user would be allowed to have a configurable number of jobs eligible to run (queued).
- Each user would be allowed to have a configurable number of jobs submitted (pending), but not yet eligible to run.
- Each state would be managed as FIFO.

However, if not enough jobs exist and the system will take idle time, the rules for selecting jobs and moving jobs between the states would expand to prevent idle.

A batch job would move from being *pending*, to being able to be selected for running (*queued*), to actually *running*. These queue states provide a way for a user who has no running or queued jobs to move ahead of another user's pending job when the other user already has queued jobs. It also provides a way for users to stack jobs for weekend runs so that the system doesn't run out of jobs.

2.13 New Utilities

As part of the new scheduler, new utilities have been developed.

2.13.1 Job Reordering Tool

The user community has been requesting a tool that would allow them to reorder their jobs. A new tool, `qorder`, was created to provide the ability to reorder jobs. The reordering preserves queue locations, but moves requests into new positions.

`qorder` works by taking advantage of an external scheduler that can process a reordering request. When the scheduler receives a reorder request, all the users jobs in the particular queue are found. A link list is used to manipulate submit times to perform the reordering. Internally to the `nqsdaemon`, the modify request routines were modified to support a new limit. `qorder` uses the `inter` routines for sending a `PKT_MODREQ` packet to the `nqsdaemon`. The packet contains the new time as an `interw32i` parameter. In order for the reordering to fully take affect, it was necessary to call `bsc_spawn()` with a new level. The new level for `bsc_spawn()` caused a scan of the queues and a recalculation of priorities without performing job initiation. By looping through all queued requests for that user, the reordering could be completed.

2.13.2 Resource Utilization Report

A program has been developed which produces a report of all batch jobs processed. In the report, the requested limits for memory, cpu time, and SRFS are displayed along with values for what was actually used. This provides the administrators with the necessary information for improving jobs and system tuning.

2.13.3 bstop

Since the scheduler runs as a system daemon, a clean way of terminating it is necessary. The program `bstop` sends a shutdown request to the scheduler so it can exit cleanly. Another way of terminating the scheduler is to send it a `SIGSHUTDN` signal.

3 New Design

The new batch scheduler will be based on categorizing a job into one of three states: *pending*, *queued* and *running*. As jobs are submitted they enter the *pending* state. Jobs will move to the *queued* state provided the users job will fit into configurable limits. The batch system will select jobs from the *queued* state to run.

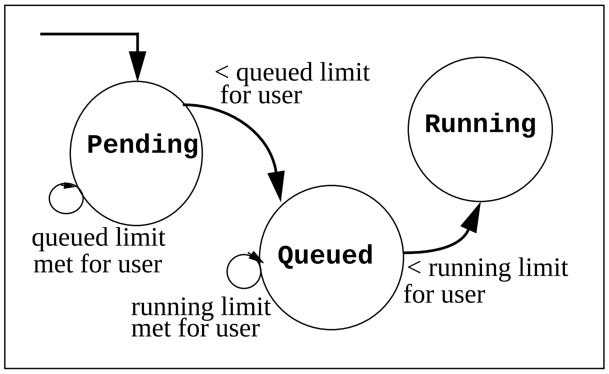

Figure 3

There are two components to the the new batch scheduler. The first is built into NQS. This component controls the flow of jobs from *pending* to *queued*. The second component is a separate process which constantly monitors and evaluates the system. It will then determine if it is safe to start another batch job. If it is safe to start another job, it will select jobs from the *queued* state.

3.1 NQS Priority

The NQS input queue priority will be modified to include additional resources to be taken into consideration when setting a jobs priority.

3.1.1 Existing NQS Priority

The basic scheduling priority algorithm for NQS as released by Cray Research is based on the time in queue, requested cpu time, fair share, and requested memory. Currently on both C90 systems, the requested cpu time, requested memory, and fair share parameters to the equation are disabled in the priority calculation. This means that the priority is based on the length of time the job is in the queue, effectively FIFO on a per queue basis.

```
 priority =
((now - state_time) * Time_wt) -
(rcpu * Cpu_wt) - (rmem * Memory_wt)
```

Fair share usage is calculated into the priority at a separate stage in the calculation.

The parameters of the equation are:

`now`	Current time in seconds.
`state_time`	Time in seconds when the job was queued.
`Time_wt`	Weighting factor for time in queue.
`rcpu`	Requested CPU time.
`Cpu_wt`	Requested CPU time weighting factor.
`rmem`	Requested memory.
`Memory_wt`	Requested memory weighting factor.

3.1.2 Modified NQS Priority

A request enters the pending state upon submission to NQS with `qsub`. The request will then migrate to the queued state and then finally begin execution.

Each *queued* job will be assigned a valid intra-queue priority based on weighting factors assigned to system resources. The existing priority equation will be expanded to include additional resources in the priority calculation. These additional resources would include the number of requested cpus' and special job status. The memory, number of cpu's, and special weights are used to give a higher priority to larger memory, multi-tasking, or special high priority jobs.

```
 priority =
((now - state_time) * Time_wt) -
(rcpu * Cpu_wt) + (rmem * Memory_wt) +
(ncpus * Ncpu_wt) +
(special * Special_wt))
```

The parameters of the equation are:

`now`	Current time in seconds.
`state_time`	Time in seconds when the job was queued.
`Time_wt`	Time in queue weighting factor.
`rcpu`	Requested CPU time.
`Cpu_wt`	Requested CPU time weighting factor.
`rmem`	Requested memory.
`Memory_wt`	Requested memory weighting factor.
`ncpus`	Requested number of CPU's.
`Ncpu_wt`	Requested number of CPU's weighting factor.
`special`	Job requires special access.
`Special_wt`	Special jobs weighting factor.

Additional weighting factors can be added to the equation as they are identified. Each factor must be a positive integer between `0` and `1000000`. Setting a factor to `0` turns off weighting for that resource.

The existing factors can be set through `qmgr` with the `SET SChed_factor` command. This interface has been expanded to permit other resource factors to be set through the `qmgr` interfaces. The new `qmgr` commands needed to implement this are:

```
SET SChed_factor Ncpu
SET SChed_factor High_priority
```

To accomplish this, two packet types were added. These are `PKT_SETSCHPRIWT` and `PKT_SETSCHNCPUWT`. Appropriate functions were added to `qmgr` to send these packets to the `nqsdaemon`.

3.2 Pending Jobs

As jobs are submitted to NQS via `qsub`, they enter the *pending* state. The production queues batch and mtask, are part of a queue complex. *Pending* jobs are held up on queue complex level limits before they can proceed to the *queued* state. *Pending* jobs are identified by a status of `Qc*` where `*` is a letter representing the limit that has been reached. In place of the priority number, *pending* jobs will have `0` displayed for their priority.

For each resource being limited in the queued state, a new status is required.

<u>Status</u>	<u>Description</u>
`QcM`	Memory limit reached
`QcT`	CPU time limit reached
`QcJ`	Jobs limit reached.

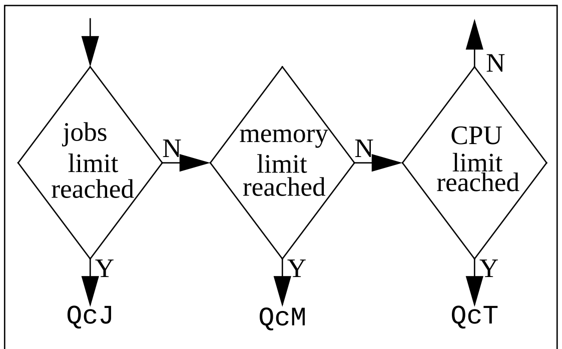

Figure 4.0, Complex User Limits

Figure 4.0 shows how each status for the resource limits is obtained. A job must pass the check for each resource before it is permitted to enter the *queued* state.

The implementation of Complex User Limits was accomplished by adding a function call to a new function `complex_user_limits()`, in the module `nqs_bsc.c` to the function `complex_limits()`. The logic for the new limits is to scan the input for every queue and build a linked list in priority order for all jobs for a particular user. For all jobs not passing the limits, the submit time is set to the current time. This forces all jobs being held in the *pending* state to appear at the end of the input queue. As jobs are released into the *queued* state, their priority will increase over time.

3.3 Queued Jobs

Jobs move from the *pending* state to the *queued* state according to predetermined criteria.

3.3.1 Limiting Resources

A user is permitted to have as many jobs in the *queued* state that would keep the user from exceeding configurable limits. Each user has the same amount of resources in the *queued* state. Resource limits are placed on memory, CPU time, and the number of jobs in the *queued* state. These limits provide the ability of keeping a single user from dominating the queues as well as to increase fairness between users.

Three new limits were required in order to allow `qmgr` the ability to set the complex user limits. These limits are:

`NQS_LIM_USERMEM`	Set complex user memory resource limit
`NQS_LIM_USERCPU`	Set complex user CPU resource limit
`NQS_LIM_USERJOB`	Set complex user job resource limit

The addition of these limits allowed for the use of the functions `setcomlim()` and `setcomquota()` for setting the limits. By utilizing existing routines, modifications were minimized.

`qmgr` is used as the interface for setting any of these configurable limits. Three new `qmgr` commands were required for setting the new complex level limits. These are:

```
SET COMplex USER_Cpu_limit
SET COMplex USER_Job_limit
SET COMplex USER_Memory_limit
```

In addition to adding new qmgr commands, one command needed to be changed due to uniqueness conflicts with the new commands. `SET COMplex User_limit` was changed to `SET COMplex USER_Limit`.

3.4 Additional Changes

In order to support the new features being added to NQS, both qsub and qmgr commands for NQS needed to be updated.

3.4.1 qsub

Three changes were required to `qsub` in order to support the new scheduling environment.

1. Number of CPU's.
2. High priority jobs.
3. Default job limits.

The first new parameter is for specifying the number of cpus required for a multitasking job. A new command line parameter, `-n` *`ncpu`*, was added so that a user can specify the number of cpus required for the job. This is used for scheduling purposes only. The possibility exists that the user may specify a different value on the command line then the user does in their scripts.

A second paramter was added for high priority jobs. The command line parameter, *`-H`*, will signify that the job is high priority. Additional logic is required to perform some authorization checking to make sure the user is allowed to submit high priority jobs. If a user specifies *`-H`* and is not authorized, the job is rejected at the time of the *`qsub`*. If this feature acceptably provides the full capability of high priority jobs, the special queues can be eliminated. High priority jobs would now be scheduled to run. When they are submitted, they will move to the top of the queue and begin running when resources are available. Currently, special jobs start immediately and can adversely affect the system.

If a user fails to specify either per process or per request limits for cpu and memory, the job receives a default limit of 4MW and 300 seconds. If per process limits are provided, they are used to set the per request limits. This feature is to encourage the specification of exact per request limits for batch jobs.

3.4.2 qmgr

All configurable parameters are set or changed using `qmgr`. As previously discussed, new `qmgr` commands were needed to set/change the complex level limits and to
set/change new weighting factors.

An additional change was required for the `snap` command of `qmgr`. This command provides the mechanism for taking a snapshot of the existing configuration of NQS. The new limits are included in the output produced by the `snap` command.

4 The Batch Scheduler

A separate program running as a system daemon is used to actually initiate jobs. This program will monitor the systems utilization and select jobs for initiation. After selecting a job to run, the scheduler utilizes the `schedule request now` functionality built into the `nqsdaemon` to initiate a job.

By using a separate daemon for initiating jobs, the ability to react to changes in system resources quickly is achieved. An example of this would be the ability to adjust to the increase or decrease in interactive load.

4.1 System Resources

Several system resources need to monitored to not over commit systems resources which would adversly affect overall system performance.

4.1.1 Swap Rate

The swap rate will be based on the last `sar` interval for the calculation of the rate. The swap rate is determined by the number of blocks swapped in per second. Swap rate is obtained from the `si.bswapin` field of the structure `sa`, where `si` is the structure `sysinfo`.

Original efforts attempted to utilize the raw data file to obtain system information. Timing problems occurred when reads to the `sar` file occurred during `sar` interval updates. The raw reads were backed out and a forked child was used to issue a `sar` command and return the information back to the schedule via a pipe Then by parsing the strings, the information could be obtained. An example of using a pipe to retrieve `sar` information would be:

```
sprintf(cmd,"sar | tail -3 | head -1");
pd = popen(cmd,"r");
fscanf(pd,"%s %d %d %d %d %d",
tstamp,&usr,&sys,&wsem,&locks,&idl);
pclose(pd);
```

4.1.2 Load Average

The definition of load average is the number of processes on the run queue and those waiting for I/O. On the Cray, all active processes have a process status of `SRUN` and those that are connected to a cpu have a process status of `SONPROC`. For the load average calculation, the sum of `SRUN` and `SONPROC` processes will be used. This information is easily obtained by scanning the `proc` table. The following example shows how this information can be gathered.

```
tabinfo(PROCTAB, &tinfo);
tsize = tinfo.head +
       (tinfo.ent *tinfo.len);
tloc = (char *) malloc(tsize);
tabread(PROCTAB, tloc, tsize, 0);
p = (struct proc *)tloc;
for(x=0;x<tinfo.ent;x++) {
   if((p->p_stat == SRUN) ||
      (p->p_stat == SONPROC))
      runq++;
   p++;
}
free(tloc);
```

4.1.3 System Time

The system time calculation will be based on the last `sar` interval. This yields the average percentage of system time across all cpu's over the last `sar` interval. System time is calculated from the `percpu[x].unixc` field of the structure `sa`. The variable `x` represents individual cpus, so a structure `percpu` exists for each cpu in the system.

As described in section 4.1.1, problems were encounted when attempting to utilize the raw sar data file. Therefore a child process communicating through a pipe was used as described in section 4.1.1.

4.1.4 Idle Time

The idle time calculation will be based on the last `sar` interval. This yields the average percentage of idle time across all cpu's over the last `sar` interval. Two classifications of idle time are used, long and short. Long idle reflects idle time of time. Short idle represents that current snapshot of the system. Idle time is calculated from the `percpu[x].idlec` field of the structure `sa`. The variable x represents individual cpus, so a structure `percpu` exists for each cpu in the system. Short idle is calcualted by looking at the current processes in the system and evaluating total processor utilization.

As described in section 4.1.1, problems were encounted when attempting to utilize the raw sar data file. Therefore a chile process communicating through a pipe was used as described in section 4.1.1.

4.1.5 Free Memory

The free memory calculation will be based on the last `sar` interval. This provides the average amount of free memory. The calculation of free memory is the total user memory minus the total user memory used minus the total memory locked. The structure `sa` contains all three items.

Field	Description
`usrmem`	total user memory
`si.umemused`	total user memory used
`si.memlock`	total user memory locked

As described in section 4.1.1, problems were encounted when attempting to utilize the raw sar data file. Therefore a child process communicating through a pipe was used as described in section 4.1.1.

4.2 Scheduling

The first step is to check NQS. If NQS is not running, then the scheduler should wait for NQS to start up. A second situation to check for is whether NQS is up and whether jobs should be started. The situation exists where NQS is up but we don't want to process user jobs. The batch scheduler will not initate any jobs in any queue that is stopped. Therefore, stopping a batch queue stops any new job from being scheduled to run in that queue. Starting the queues will cause the scheduler to resume normal operation.

The queues are configured as running and enabled but with a run limit of 0. This allows NQS to start all checkpointed jobs first at startup. This functionality serves two purposes. The first is that all jobs that were running continue to run when the system is restarted. Second, the system can become loaded in a much shorter time period at system startup.

Deferred jobs are run only when there exist no jobs in the queued state that can be started. The scheduler will consider all jobs in the queue state before considering any deferred job. Also, deferred jobs will only be started if idle time will be accrued. If the systems resources can support another job but the cpus are all 100% busy, a deferred job will not be started.

The scheduler takes a different approach to counting batch jobs. Rather than actually counting the number of running jobs, it is designed count the number of cpus required. This allows for multi-tasking jobs to be counted based on the number of cpus required. When looking at what's running, it's not the number of jobs but rather the number of cpus that's of concern. A 16 processor job has a greater impact than a 1 processor job. Since this required the addition of a new command line parameter, an incompatibility with standard NQS's was revealed. A method of adding a new script processed arguement similar to `#QSUB` is under investigation.

4.3 Initiating Jobs

There are two methods that can be used to start batch jobs. The first is fork a process that executes the `qmgr` command. The second is to build the ability to start another job into the scheduler. This second method was chosen. The idea is to provide a mechanism for the scheduler to issue the command packet directly to the `nqsdaemon`. By utilizing the NQS libraries, it was possible to use the function `schedreq()` with the `SCHED_NOW` option. The return code is then examined to check job initiation status.

However, there is a side effect to using the NQS library functions. The library routines open the `nqsdaemon` request pipe and leave it open. This means that NQS cannot be shutdown until the scheduler is shutdown.

5 Batch Resource Accounting

In order to track what a user requests versus what a user actually uses, a "batch resource accounting" file is maintained. When a batch job terminates, a record is written that contains all the limits the user requested as well as what the user actually used. This will allow us to assist the users with tuning their jobs to run more efficiently in the system. For example, if a user requests 900MW of /big but only uses 100MW, then the user can be contacted and the request reduced. This frees limited resources for other users to utilize.

When a users job terminates, a report on the users resource requirements is automatically produced at the end of the stdout file. This would allow us to take a pro-active approach to helping the user adjust resource requirements appropriately.

6 Lessons Learned

As development progressed, it became apparent that, although the user exits are a good feature, they weren't sufficient. More user exits are needed and more flexibility is needed with exiting ones.

Although the functionality added to NQS is complicated, the implementation is not complex. Porting to new releases is an easy task to perform. However, more enhancements to improve performance and to add greater flexibility could be useful.

The user community has also received a benefit from this work. Not only has the batch environment been greatly simplified, but more power has been given to the user to control their own jobs. The information is always supplied to the user about resource utilization for a job. This in turn allows the user to more efficiently submit their job.

By getting accurate information about all jobs, throughput can be increased for all size jobs. The system resources are fully utilized by taking advantage of what's available.

7 Summary

The enhancements made have simplified the queue structure eliminating confusion amongst the user community. Additionally, by using requested limits as opposed to queue limits for scheduling, the mainframes can be more efficiently used. The new command, `qorder`, was welcomed by the users. The enhancements made have produced positive results to date.