FORSCHUNGSZENTRUM JÜLICH GmbH
Zentralinstitut für Angewandte Mathematik
D-52425 Jülich, Tel. (02461) 61-6402

Interner Bericht

Automatic Checkpointing of NQS Batch Jobs on CRAY UNICOS Systems

Norbert Attig, Volker Sander

KFA-ZAM-IB-9303

März 1993
(Stand 24.03.93)

Erscheint in Proceedings Cray User Group Meeting (Spring), 1993, Montreux

Automatic Checkpointing of NQS Batch Jobs on CRAY UNICOS Systems

N. Attig and V. Sander

Zentralinstitut für Angewandte Mathematik
Forschungszentrum Jülich GmbH (KFA)
Postfach 1913, 5170 Jülich, Germany

Abstract

In most UNIX systems long running application programs are not protected against the loss of their accumulated CPU time in case of regular shutdowns or system crashes. In contrast to these systems, the UNICOS operating system provides a *checkpoint/restart* facility, which allows e.g. to recover NQS batch jobs after a regular system shutdown and reboot. However, there is still no function, which periodically performs checkpointing of running processes. This kind of checkpointing, which would minimize CPU time losses in case of system crashes, is completely left to the user. Unfortunately, most of the users do not care about checkpointing. Therefore, a feature was developed at KFA, allowing to checkpoint NQS batch jobs automatically after a certain CPU time interval. The key issue of this feature is a UNIX daemon which is activated together with each NQS request. We present a detailed description of the daemon and its user interface. Our experience in a production environment shows, that the CPU time losses due to system crashes can be drastically reduced by this feature.

1. Introduction

Generally, the problem of loosing accumulated CPU time in case of system interrupts is not solved in UNIX systems. This may be tolerated on small personal workstations but cannot be accepted on high-end supercomputers where CPU time is extremely valuable. In order to make programs recoverable after a (regular) shutdown or a system crash, the UNICOS operating system offers a mechanism called *checkpointing* to create a *restart file* of any process or job at any time. This can be done either by the system call *chkpnt* or, especially for NQS (network queueing system) batch jobs, by the user command *qchkpnt*. If a system interrupt happens, the restart files are used to recover processes as well as jobs after the reboot. E.g. NQS uses this mechanism to write a restart file of each job when it receives a shutdown signal. However, there is no function, writing restart files of running processes in certain CPU time intervals.

At KFA two CRAY systems are installed, a CRAY Y-MP8/832 and a CRAY Y-MP M94/4256 (formerly a CRAY X-MP/416). Both machines are operated in batch mode and fully utilized most of the time. The maximum time given to an NQS batch job at our site is limited to 10 CPU hours, and in most of the jobs this maximum is specified. In spite of the fact that the hardware is quite stable and system crashes do not occur very often, we want to minimize the loss of CPU time in case of sudden system interrupts. Unfortunately, at our site only a few users write restart files of their programs explicitly. Therefore, we investigated the possibility to write restart files of running jobs automatically. Due to our NQS batch job environment we concentrated on NQS requests only. The terms “job” or “batch job” are in the following always synonyms for “NQS request”. The idea of checkpointing NQS requests can be accomplished in different ways. A brute force method would be to shutdown and restart the NQS every n minutes. But this method would have several disadvantages:

- An NQS shutdown can take several minutes. During this time the system utilization is quite bad (swap rate and I/O overhead increase strongly, processors get idle).
- Jobs with only a few seconds accumulated CPU time in the last *n* minutes are checkpointed.
- If the user performs user-triggered checkpointing, the restart files written by this method are useless.

We decided to develop a checkpoint daemon, invoked by a global profile (*/etc/profile* or */etc/cshrc*). Each batch job is then controlled by its own daemon. This daemon writes a restart file of the job after every *n* minutes CPU time used. This procedure avoids all the disadvantages of the brute force method described above. Each job is checkpointed individually after accumulating *n* minutes CPU time. Therefore, no useless restart files are written. The system utilization is no longer affected by a global checkpointing of all NQS jobs, because now the checkpointing is spread over the time. Furthermore, a user interface to the daemon can be supplied to include user-triggered checkpointing. From our point of view a user must have the possibility

- to stop automatic checkpointing,
- to change the interval time between two checkpoints and
- to switch to user-triggered checkpointing.

In the next chapters the details of our feature are discussed.

2. The Users View

A new feature will only be accepted by the users if it has obvious advantages, is easy to use and the costs to pay for using it are negligible. Let's see whether the checkpointing feature matches these criteria.

The default is, at least at our site, that most of the CRAY users do not protect their jobs against emergency system crashes. These users already take profit of the feature by doing nothing specific. Each job executes in the beginning either */etc/profile* or */etc/cshrc*. In these profiles our checkpoint daemon is started. If the user does not manipulate the automatic checkpointing, a restart file of the job will be written automatically every 30 minutes CPU time used. A log message of each checkpointing activity is written to the job's standard error file. The CPU time overhead induced by the feature is very small. A NQS job running 10 CPU hours and using the automatic checkpointing with default interval time of 30 minutes between two checkpoints, typically has to pay for additional 240 milliseconds CPU time. Statistics show that after an emergency system crash 85% of the running NQS jobs can recover from their latest restart file (created by our feature).

In addition to the automatic checkpointing facility running without intervention, the user has multiple choices to trigger our checkpoint daemon. Some users may want to change the time interval between two checkpoints from the default of 30 minutes to a larger or a smaller value. From the daemon's mode of operation, discussed in the next section, it will become clear that especially for jobs running in multitasked mode a smaller value makes sense. The user can change this interval by the command *kfa.new_chkpt n* where *n* is the time in seconds (the minimum is set to 35 seconds).

Only very few users are really interested in doing user-triggered checkpointing. They may have experienced that sometimes their jobs can not recover from the last restart file after a sudden system crash. What are the reasons which prevent a successful job recovery? It must be mentioned that the restart ability depends on several conditions. Most of them can only be ensured by the system administrators but a few can be influenced by the user. Jobs which access datasets via NFS (network file system), deal with tapes or perform non-sequential I/O are in general not recoverable. Quite often the following case prevents a successful restart: An application program writes to a file, rewinds the file and continues writing to this file. Then it can happen, that the file size at checkpointing time is larger than the file size in the moment of the system crash. In such situations, a job cannot be restarted because probably wrong results would occur. This situation can easily be prevented by the user. Each time the file has been rewound, a dummy variable must be written to the file, the file has to be closed (to flush the library and system buffers) and finally a user-triggered checkpoint has to be performed.

User-triggered checkpointing can be switched on by the command *kfa.own_chkpt*. This command disables the automatic checkpointing. Furthermore, a module called *WRCHKPT.o* has to be added to the list of object files in the loader command. Finally, one has to insert checkpoint calls (*CALL WRCHKPT()*) at proper source code positions. For example:

```
# QSUB directives
   ...
kfa.own_chkpt
cat > prog.f << 'EOF'
      PROGRAM TEST
      ...
      ...
      CALL WRCHKPT()
      ...
      END
EOF
cft77 prog.f
segldr prog.o /usr/local/lib/WRCHKPT.o
./a.out
```

There is a second way to perform user-triggered checkpointing. In that case, one has to stop our checkpointing daemon by the command *kfa.stop_chkpt* and to make use of the special purpose interface routine *ISHELL*, which passes its argument as a command to the Bourne-shell. The current process waits until the shell has completed.

```
# QSUB directives
   ...
kfa.stop_chkpt
cat > prog.f << 'EOF'
      PROGRAM TEST
      ...
      ...
      ISTAT=ISHELL('qchkpnt')
      ...
      END
EOF
cft77 prog.f
segldr prog.o
./a.out
```

There were mainly two reasons for developing our own user-triggered checkpointing performed by the routine *WRCHKPT*, both based on the functionality of *ISHELL*: The first reason was that the *ISHELL* routine implemented in UNICOS 5.0 could not be executed within multitasked programs, which is possible now. The second reason was that in former UNICOS releases *ISHELL* performed a *fork* (resp. *tfork*) system call of the calling process. The result was that the calling process *a.out*, which normally has a large memory requirement, had to be doubled first, then its forked child process was superposed by the shell. This caused a tremendous overhead. Since UNICOS 7.0, *ISHELL* uses the system call *vfork* which drastically reduces the memory usage.

3. Daemon Internals

The key issue of our checkpoint feature is the checkpoint daemon itself. It is called *qchkpt_daemon* and is forked by the command *kfa.start_chkpt n* (*n* is the time between two checkpoints in seconds), which is integrated in the global profiles (*/etc/profile* and */etc/cshrc*). The main part of the daemon is nothing else than a wait loop, where the daemon is waiting for arriving signals (see Fig.1). In this loop different functions will be performed, if special signals arrive. These signals are the alarm signal (SIGALRM (14)) and four user-defined signals (SIG_FOR_NEW (60), SIG_FOR_END (61), SIG_FOR_USER (62), SIG_FOR_MAKE_OWN (63)). The last four are sent by the commands *kfa.new_chkpt*, *kfa.stop_chkpt*, *kfa.own_chkpt* and by the subroutine *WRCHKPT*, which performs user-triggered checkpointing directly out of programs. Communication between the daemon and some of the commands is accomplished by named pipes, created in the job's temporary directory *$TMPDIR*. The commands identify "their" daemon by its process-id, which is picked up from the job's process table. If the user has not switched to user-triggered checkpointing, every *wait_time* seconds (initially *wait_time* is set to the given time interval between two checkpoints) a SIGALRM signal is sent by the daemon to itself via the system call *alarm*. The daemon catches this signal by the system call *signal*, calculates a new *time_till_write* and then checks whether the job has to be checkpointed. Finally, *time_till_write* updates *wait_time*. In case of user-triggered checkpointing, the daemon does nothing until signalled explicitly. The main functions of the daemon are now discussed in detail:

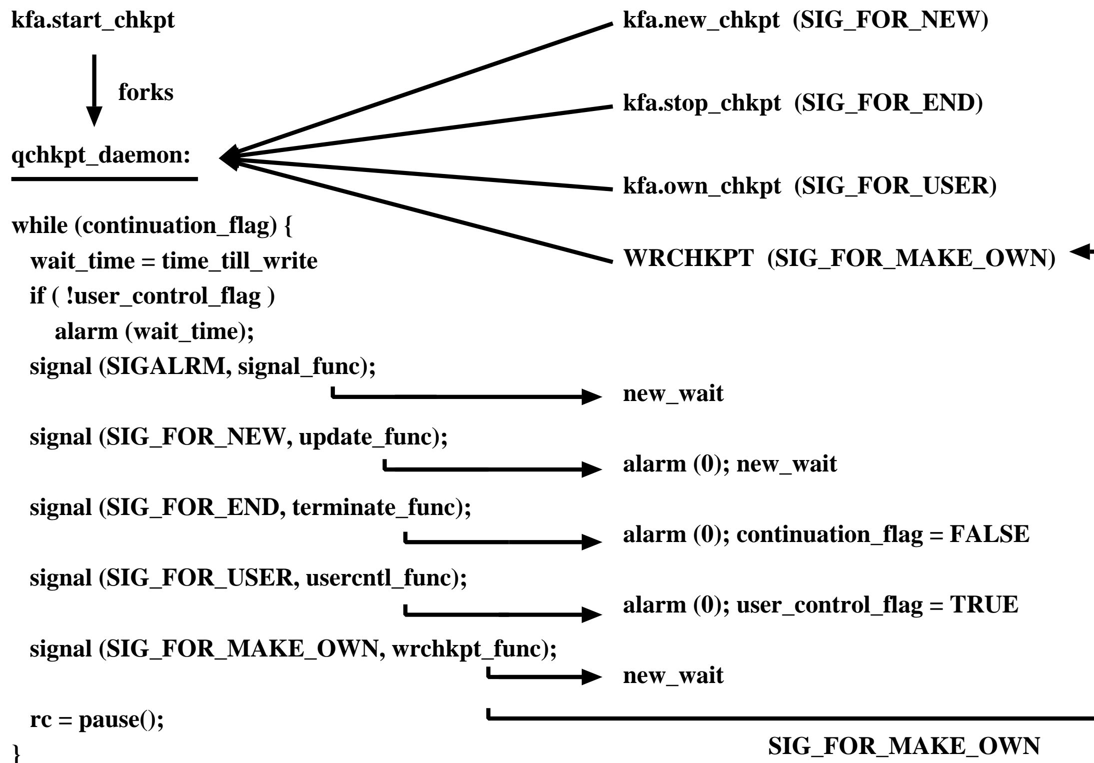

Figure 1. The checkpoint daemon *qchkpt_daemon*

signal_func

This function is performed in case of a SIGALRM. Based on the routine *new_wait* it is determined whether a restart file has to be written or not.

update_func

Receiving a SIG_FOR_NEW, the daemon executes this routine. It communicates with the client (*kfa.new_chkpt*) via a named pipe to get the new interval time between two checkpoints. The system call *alarm(0)* is executed to cancel the previous alarm request. At the end, *new_wait* is called to evaluate the new time to sleep and, eventually, to checkpoint the job.

terminate_func

A shutdown of the daemon in an orderly fashion is signalled by SIG_FOR_END. This causes the daemon to execute the function *terminate_func*, where *alarm(0)* cancels the previous alarm request and the external variable *continuation_flag* is set to FALSE. Consequently, the while-loop of the daemon has to be finished.

usercntl_func

If the user wants to switch to user-triggered checkpointing, he has to execute the command *kfa.own_chkpt*, sending the signal SIG_FOR_USER to the daemon. The daemon processes this signal by executing the function *usercntl_func*, setting the *user_control_flag* to TRUE. This disables automatic checkpointing. The job is now only checkpointed by the daemon, if the daemon is explicitly signalled by the subroutine *WRCHKPT*.

wrchkpt_func

This function is performed by the daemon in case of receiving a SIG_FOR_MAKE_OWN. It reads from a named pipe the process-id of the calling process (the program which has called *WRCHKPT*), performs a checkpoint of the whole job by means of the function *new_wait* and finally sends back the signal SIG_FOR_MAKE_OWN to the calling process, using the process-id read in the beginning. Not until after receiving this signal the subroutine *WRCHKPT* returns and the executable continues running. The handshake guarantees that the checkpoint is written just in time.

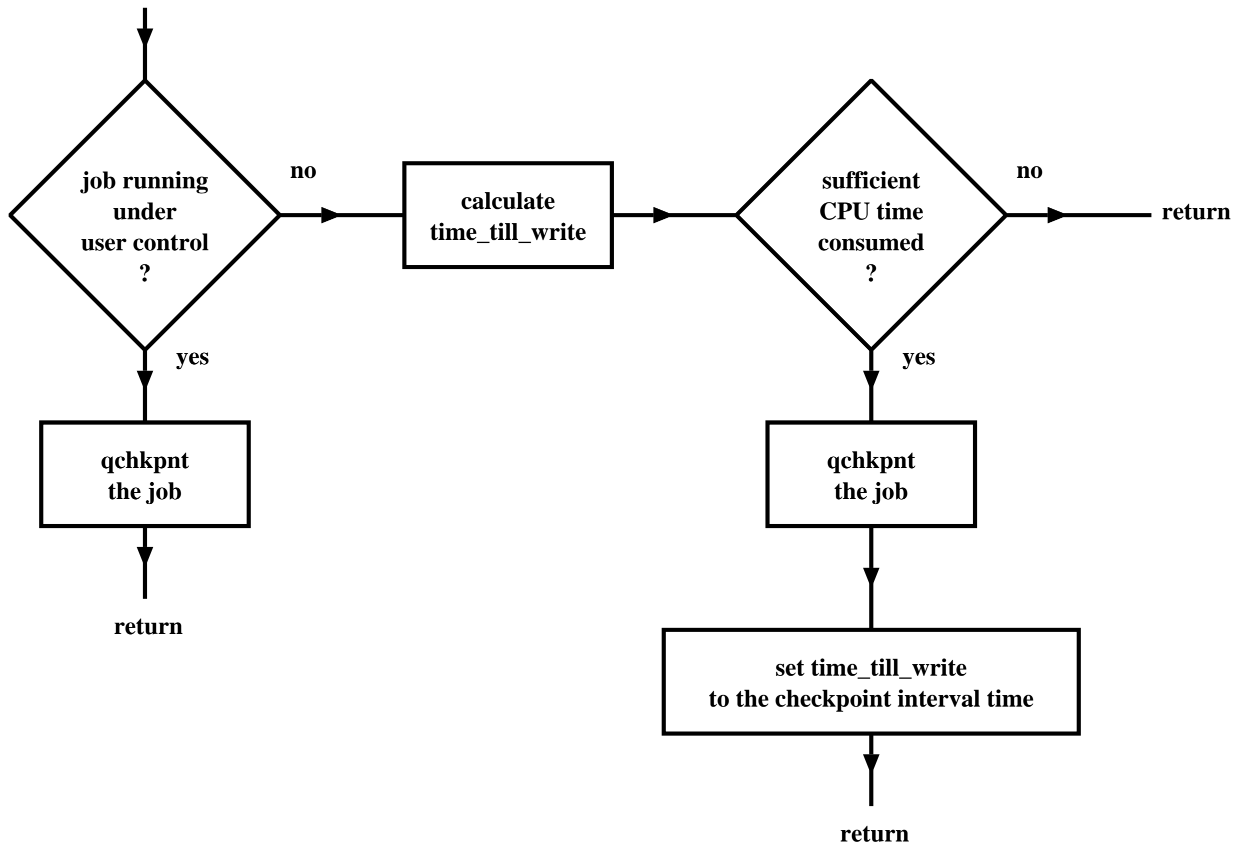

Figure 2. The central checkpointing function *new_wait*

From the descriptions above it is easy to see that the checkpointing is effectively done by the function *new_wait* (see also Fig.2):

new_wait

This function checks first whether the *qchkpt_daemon* operates under user control or automatically. If it operates under user control, the *qchkpnt* command is forked and executed immediately, which means the job becomes checkpointed. A log message of this activity is written to the job's standard error file. If the daemon operates automatically, the next action is to determine the global variable *time_till_write*. This variable specifies the difference between the used CPU time (user and system) of the job since the last checkpoint (or the start-up of the job) and the time interval between two checkpoints (normally 30 minutes). If this difference is smaller than 10% of the checkpoint interval time, the checkpoint will be performed, a log message of this activity will be written to the job's standard error file and the variable *time_till_write* will be reset to the checkpoint interval time. Otherwise the function returns without checkpointing. The reason for doing the automatic checkpoint within a 10% interval is easy to understand. It is necessary to avoid frequent calls of *new_wait* when the accumulated CPU time is close to the checkpoint interval time. The variable *time_till_write* is used by the daemon to update its variable *wait_time*, which determines the next sending of a SIGALRM signal.

A detailed description of the user commands *kfa.start_chkpt*, *kfa.new_chkpt*, *kfa.stop_chkpt*, *kfa.own_chkpt* and the subroutine *WRCHKPT.c* can be omitted at this point, because their functionality is explained above.

Finally one remark on the shutdown procedure of the checkpoint daemon: The daemon can be stopped regularly by the command *kfa.stop_chkpt* or abruptly by the end of the NQS job. It does not matter which alternative is used.

4. Experiences and Summary

At our site, the automatic checkpointing feature has been running in production mode for quite a long time (since 1989) and our experiences have been very good. The CPU time losses due to system crashes have been reduced drastically. Without our feature, NQS jobs could only be recovered after a system crash from the very beginning or from restart files written by the last NQS shutdown, because 99% of our users do not care about checkpointing. We observe that with our feature more than 85% of the running jobs can recover after a crash from their latest restart file written by the automatic checkpointing feature. This saves a large amount of accumulated CPU time and is a significant improvement for our users.

The CPU time overhead induced by the additional checkpointing activity is negligible. It depends on the memory usage of the job to estimate the CPU time needed for a restart file creation. Our statistics show that it takes less than 10 milliseconds on the average for our job profile. The CPU time usage of the checkpoint daemon depends on the time interval of two checkpoints. Using the default of 30 minutes and considering a normal production batch job, which consumes 10 CPU hours, the daemon activity takes 40 milliseconds on the average. Therefore, the total CPU time spent for the checkpointing feature in a typical production job is about 240 milliseconds. Of course, checkpointing induces also I/O overhead and disk activity but this does not affect system utilization very strongly because the checkpointing activity is spread over the time.

From the systems point of view, no additional disk space needs to be allocated for the restart files created by our feature. They are written to */usr/spool/nqs/private/root/chkpnt*, the default checkpoint directory of NQS, where sufficient disk space is available. Of course, only the latest restart file of each job is stored.