REMOTE UNIX
TURNING IDLE WORKSTATIONS INTO CYCLE SERVERS

Michael J. Litzkow

University of Wisconsin
Computer Sciences Department
Project Topaz
ucbvax!uwvax!mike or mike@wisc.edu

ABSTRACT

A computing environment consisting of workstations connected by a local area network is now common. Often these workstations are assigned to individual users, and thus represent a significant unused resource when those individuals are not working. Remote Unix, (RU) uses these idle workstations to execute compute-bound jobs in the background. Users submit jobs to RU from their own workstations. The jobs are queued, and eventually executed remotely on idle workstations. When their jobs have completed, users are notified by mail. The owner of a workstation has absolute priority over RU jobs. When the owner initiates interactive or other non-RU work, any RU job is automatically checkpointed and restarted on another workstation. RU jobs may go through any number of checkpoints before eventual completion. Finding idle workstations, handling of remote system calls, and checkpointing when necessary are all handled by the RU software without intervention from users. In addition to providing ‘‘free’’ cycles, RU allows completion of very long running jobs which might otherwise be aborted by system crashes and shutdowns. The longest running job so far has completed successfully after accumulating 60 CPU days over a 3 month period. This paper describes the computing environment for which RU was designed, and current limitations on the class of jobs which it can handle. The three main components of RU — remote system call handling, a general UNIX† checkpointing facility, and distributed spooling and control — are each discussed. The current version of RU supports only single process jobs. Possible extensions are discussed.

1. Introduction

The advent of workstations and local area networks has had a dramatic impact on productivity in the UNIX environment over the past several years. Unfortunately, many of these same workstations which are so productive during the work day, are often completely wasted at other times. Remote Unix is a facility which allows these "wasted" cycles to be used productively by compute-bound jobs.

1.1. Our Local Environment

The University of Wisconsin Computer Sciences department currently has about 100 MicroVaxII†† workstations, several VAX 11/750’s††, and two VAX 11/780’s††. All are connected by a local area network, run the same version of UNIX, and are object-code compatible. The majority of the workstations are placed in individual’s offices, and are dedicated to the use of that individual, (the "owner"). Owners expect to have full use of their workstation at any time, day or night. Most usage is done in the owner’s office, but occasionally users log in from other machines or dial up from home.

We also have a number of users who are interested in running large, compute bound jobs (batch jobs), such as simulation programs. Often a user will want to run many copies of a long-running simulation with slightly different parameters.

We have developed the Remote Unix (RU) facility to accommodate batch jobs by using the otherwise idle workstations in a way which does not interfere with the normal work of the workstation’s owner.

†UNIX is a trademark of AT&T Bell Laboratories

††MicroVaxII, VAX 11/750, and VAX 11/780 are trademarks of Digital Equipment Corporation

The RU package consists of facilities for queuing and execution control, automatic checkpointing, and remote execution of batch jobs. Users submit jobs to the RU queue, and when some workstation becomes "idle", the job is remotely executed on the idle workstation. Most system calls are done remotely, so that file accesses and user ID's refer to the environment in which the job was submitted. When the remote workstation's owner wants to regain use of the workstation, the job is automatically checkpointed to a file on the originating machine. When some other workstation becomes idle, execution can resume. The system takes care of re-opening all the files in the correct modes, assigning the original file descriptor numbers to them, and seeking to the appropriate places. When the job completes, the user is sent mail regarding its completion status. Some implementation details and current limitations are discussed below. RU is implemented entirely through special libraries and user level programs; it does not depend on any modifications of the UNIX kernel.

2. Remote Execution

RU jobs are linked with a special version of the C runtime library, in which most of the system call "stubs" have been replaced with remote execution code. This code marshals the arguments, sends the request over the net to a "shadow" process on the originating machine, and places the results into the correct user area(s). A few system calls (e.g. sbrk) are done locally.

When a job is to be executed remotely, a shadow process is created on the originating machine. The shadow executes a simple program called the "starter" on the remote machine using the standard C library routine rcmd(3). The starter copies the core image to a special directory on the remote machine, sets up pre-defined file descriptors as network connections to the shadow, starts execution of the job, then waits for it to exit. When the job completes, the core image on the remote machine is removed by the starter. The core image is in a format which is suitable for exec, but contains extra information regarding currently open files, contents of registers, and current working directory. The RU library contains startup code which opens the necessary files corresponding to stdin, stdout, and stderr before "main" is called.

2.1. Remote execution limitations

The current system does not support the UNIX system calls pipe(), fork(), exec(), signals, or inter-process communication. Programs which depend on knowing their own process id will not work correctly because the id of an RU process changes after each checkpoint. Programs which use "wall clock time" will not produce re-produceable results. Programs which do large amounts of I/O are not appropriate for RU, since the overhead of the remote read and write operations is prohibitive.

Despite all the limitations, there are a large number of compute intensive jobs which are well suited to RU. We have also been able to run some jobs which could not practically be run on a normal UNIX system because of the large amount of CPU time required.

3. Checkpointing

When a workstation owner resumes use of that workstation while it is executing an RU job, the job is stopped and checkpointed on the originating machine. This allows preemption of jobs without loss of work, and provides fault-tolerance. Should the system crash before an RU job is finished, the job will automatically be restarted from the latest checkpoint.

Checkpointing is accomplished via the remote system call mechanism in response to a signal. A signal handler is supplied by the RU library which catches the signal, and writes out the necessary information. The registers are saved on the stack, and the open files and other special data are stored in the program's data segment. The program's data and stack are then written to the checkpoint file via the remote write system call. The checkpoint file is written in a UNIX executable format, and the entry point is set to a routine called "restart".

A program is restarted from a checkpoint file by the "exec" system call, just like any other UNIX executable. The restart routine changes to the current working directory, re-opens the files and seeks to the appropriate places, restores the stack and registers from the checkpoint file, and continues execution from where it was suspended at the time of the checkpoint.

4. Spooling and Control

Each machine maintains its own queue of RU jobs which it wishes to run remotely. Users submit jobs via the "ru" program, giving program arguments, and possibly specifying files to be used in place of stdin, stdout, and stderr. If the user does not specify redirections for the standard files, default names are generated (e.g. /dev/null for stdin). Programs are also provided for displaying the RU queue, changing priority of programs within the queue, and removing RU jobs prior to completion.

The control system consists of a "central resource manager", which gathers general information about all the machines, and a "local scheduler" which makes decisions affecting only a particular workstation.

The resource manager periodically polls all the schedulers, determining which workstations are accepting ru jobs, which have jobs to run, and how many individual users are waiting for those jobs. Since the resource manager requires only summary information from the other machines, polling can proceed quickly and without much communication overhead. Upon finding an "idle" workstation the manager chooses another workstation which wants to run a job, and sends it permission to run a job on the idle machine. The machine which receives permission to run an RU job remotely has responsibility for choosing among the jobs it wishes to run, and communicating directly with the idle workstation to initiate execution.

Each workstation in the RU "pool" runs a program called the local scheduler. The local scheduler is responsible for determining whether or not the workstation on which it is running is "idle", and for getting rid of RU jobs when the workstation's owner returns to work. This is accomplished by a finite state machine within the scheduler. Periodically the scheduler wakes up, examines the process queue on its machine, evaluates the load, and updates the state if needed. Only jobs belonging to ordinary users, (not system processes or RU jobs), are included in this load evaluation. When the load has been low for a period of time, the scheduler determines that its machine is idle and can accept ru jobs. At this point the scheduler's state machine enters the "accept" state. When the machine acquires an RU job to run from another machine, its state becomes "RU". Only one RU job is accepted at any one time. If a user returns to the workstation and attempts to use it, any RU job which is running will be temporarily stopped. Assuming the user continues to use the workstation, the RU job will later be checkpointed back to its originating machine, and the workstation will return to the "user" state. If the user only uses the workstation for a short time, then leaves again, the RU job will be resumed, and the workstation will return to the "RU" state. The scheduler also accepts permission to run RU jobs remotely, and chooses which job to run when given permission.

A side benefit of the scheduler is a "screen saver". Whenever the workstation is "idle", i.e. ready to accept RU work or doing RU work, the screen saver is put on the workstation's monitor. The screen saver also notices any keyboard activity, and informs the scheduler that the machine is no longer idle. Thus workstation owners can just walk away from their machines at any time, leaving them available for RU, and can regain control immediately when they return.

Due to the partially distributed control system, a two level priority system is used. The resource manager implements a priority scheme based on how many separate users have jobs queued on each workstation, and how long those jobs have been waiting to run. When there are idle machines, the resource manager chooses the machine which has had the most users waiting the longest time, and gives it permission to remotely execute a job on the idle machine.

The scheduler which receives such a permission must choose a job from its local queue to be run. Each process in the queue has a priority. Users can specify what priority their jobs should have when they are submitted, but only the super user can specify a priority greater than the default. If there are more than one job at the highest priority level, the scheduler will choose the oldest one.

5. Future Work

The primary limitation of the current RU is the range of system calls supported. Eventually, we plan to implement all UNIX system calls except possibly the networking calls. We have already begun work on signal implementation, which is not too difficult. The main steps are discovering all the caught, blocked, and ignored signals, as well as any pending signals at the time of checkpointing so that they can be restored during the restart procedure. Fork can most easily be implemented by executing a fork in the shadow as well as the remote job. Thereafter each job is separate, has its own checkpoint file, etc. The simplest implementation would be to require that all processes in a "family" be loaded on a particular remote machine at the same time. Exec could be implemented by overwriting an RU job's checkpoint file with a new one, then using the UNIX exec system call to overwrite the executing core image with the new one. Pipe will require special handling at checkpoint time to "drain" the pipe and save those contents somewhere in the checkpoint file.

Another useful enhancement would be access to "local" files for certain applications. For example, font-description files are replicated on all machines, so a typesetting program should access the local copy rather than the copy on the originating machine.

Acknowledgements

This project is based on the idea of a "processor bank", which was introduced by Maurice Wilkes in connection with his work on the Cambridge Ring.[1]

I would like to thank Don Nuehengen and Tom Virgilio for their pioneering work on the remote system call implementation; Matt Mutka for first convincing me that a generalized checkpointing method could be practical and for ideas on how to distribute the control and prioritize the jobs; Marvin Solomon whose "red pen" contributed greatly to the readability of this paper; and the countless others who have contributed their help and ideas to the project.

This research was supported by the National Science Foundation under grants MCS81-05904 and DCR-8512862 and by a Digital Equipment Corporation External Research Grant.

Wilkes, M. V., Invited Keynote Address, 10th Annual Interational Symposium on Computer Architecture, June 1983.