User-Oriented Resource Scheduling in UNIX

John K. Edwards and Pei Cao
Computer Sciences Department
University of Wisconsin-Madison
Madison, WI 53706
{edwards,cao}@cs.wisc.edu

Abstract

In many current operating systems resource scheduling decisions are made exclusively based upon throughput considerations. With the trend toward personal workstations, scheduling of resources, in particular physical memory and disk access, should be more user oriented. Local modifications are made to the Linux kernel to allow the user to guarantee minimum or maximum amounts of physical memory for particular processes and to set disk access priorities for processes. A daemon that authenticates all requests to adjust resource allocation and which automatically sets default allocations and priorities for new instances of interactive programs is implemented. The daemon also dynamically adjusts limits for inactive interactive processes. Experimental results suggest appropriate minimum memory allocations for various programs. These adjustments result in significantly improved responsiveness for interactive applications, with minimal impact on the throughput of background applications.

1 Introduction

An important task of operating systems is to manage the allocation of resources to multiple processes. Many modern operating systems, however, do not manage resources in a user-friendly way. They often consider throughput alone, and ignore the relative importance of processes. The user has no control on how much resources a process can take. If a background process or other users' processes monopolize the resources of a machine, the user cannot instruct the operating system to correct it, even if he or she bought and owns the machine.

For example, if a process needs a lot of memory and starts to thrash the machine, that process is awarded with all of the machine's main memory. When this happens, the user finds that the interactive processes are frequently paged out and become sluggish to keystrokes or mouse movements. The user then feels that the usability of the machine is significantly reduced.

Modern operating systems adopt the throughput-oriented, uncoordinated approach to resource management because they were designed to run on time-sharing computing servers. As cost concerns and technological advances push PCs and workstations to replace servers, we believe it is time to reconsider how operating systems should manage resources on these platforms. The owner of a machine should be given control over resource allocation. In addition, user satisfaction should be the goal of resource management. That is, both the interactive applications' prompt responses to user inputs and the throughput of the system should be considered.

We propose a *user-oriented* resource management approach for operating systems on PCs or workstations. The approach introduces a set of simple kernel mechanisms to allow users to adjust memory allocation and I/O priorities, and an authorization scheme to prevent users from misusing the kernel mechanisms. The approach also includes a user-level policy daemon, which monitors and dynamically adjusts resources to create a better working environment for the user. In particular, it recognizes the current interactive processes and reserves resources for them. In the mean time, it lets non-interactive processes have the rest of the resources and keep progressing.

We have implemented user-oriented resource management in Linux [14]. The kernel modifications are limited to two modules and are quite simple. We found that the facility is easy to use and gives us greater control over the machine, Together with other tools like "top", the facilities provide a nice way for users to tune the performance of various processes. In addition, the user-level daemon makes the new system behave much more pleasantly in over-loaded situations. On average, reserving resources for interactive applications only slightly increases the turn-around time of non-interactive processes (less than 20% see Table 13). As a result, in our daily work we use the modified kernel on our PCs.

2 Design

User-oriented resource management consists of three components:

- kernel mechanisms to allow users to control resource allocation;
- authorization policies deciding which user can reserve or limit resources for which process;
- a user-level policy daemon that implements the authorization policy, carries out users' requests for resource allocation, monitors and dynamically adjusts resources for interactive processes.

Below we describe each component in more detail.

2.1 Kernel Mechanisms

Traditionally, the kernel controls physical memory allocation and I/O scheduling using global policies that optimize throughput. There is no user level input on the resource allocation, the rationale being that the kernel is designed for time-sharing systems and all users should be subject to the kernel's decisions so that the throughput is maximized.

As personal computers and workstations become more widespread, there emerges a need for users to be able to adjust resource allocation. Many PCs or workstations are bought by individuals and shared by a number of people; the owner may wish to make sure that his or her jobs will always receive priority in obtaining resources. Users may also want to limit resources used by background jobs in order for foreground jobs to perform better. Existing mechanisms such as mlock or mpin do not satisfy this need because they can only be used by the superuser, and they require precise information on which parts of a process's address space need to be resident in memory.

We introduce the following system calls to give users control on memory allocation and I/O scheduling:

- `min_memory(pid, size)`: asks the kernel to give process `pid` at least `size` amount of physical memory ;
- `max_memory(pid, size)`: asks the kernel to never allocate more than `size` amount of physical memory to process `pid`;
- `io_priority(pid, pri)`: tells the kernel that disk I/O requests from process `pid` have priority `pri`; lower `pri` indicates lower priority. Default priority is 0. Common `pri` values are 1 (for interactive processes) and -1 (for background processes).
- `register_resource_daemon(pid)` : Notifies the kernel that all of these system calls should only be accepted from process `pid`. All requests by other processes must be funneled through this daemon.

Existing kernels can implement these system calls quite easily. As will be seen in Section 3, implementing the calls in Linux requires only modest changes in the VM module and the disk scheduling routine.

We separate the mechanisms for user control from authorization and policy issues: the system calls can only be called from the user-level policy daemon. A user's request on resource allocation is implemented as a message to the daemon, which checks whether the user can issue the request, and issues the appropriate system calls. This way, the authorization policy can be changed easily at runtime without involving the kernel.

2.2 Authorization for Resource Control

Giving users the opportunity to manage resources does not mean any user can deprive other users of resources. The authorization policies, implemented by the policy daemon, prevent this from happening. Authorization deals with who can make what request on which process's resources. Similar to access control matrices, authorization policies can be expressed as matrices: users are rows, process types are columns, and allowed requests are matrix elements.

For the environments we are considering (i.e. PCs and workstations), we consider only two types of processes: those that are owned by the user, and those that are not. We also consider only two types of requests: reservation and limitation. Setting upper limits on physical memory and setting the I/O priority to below 0 are limitation requests; setting lower limits on physical memory and setting I/O priority to above 0 are reservation requests.

In our system we use the following simple policy. There is a special user who is considered the owner of the machine. The owner can perform all requests on all user processes, and all other users can only limit resources on their own processes. In other words, the authorization matrix is:

user	user's processes	other users' processes
owner	reserve/limit	reserve/limit
other	limit	none

Note that even though non-owner users cannot reserve resources for their processes, they can indirectly improve the performance of some processes by limiting the resources consumed by other processes.

Clearly, different environments need different flavors of authorization policy. The groups of users, types of processes and types of requests can be much more elaborate than what we described. Once the users, types of processes and requests are set, the policy can be expressed as a matrix, put in a configuration file and interpreted by the policy daemon.

2.3 User-Level Policy Daemon

The policy daemon serves two functions:

1. to enforce authorization policy and carry out legitimate requests from users;

2. to improve the user's working environment by identifying the processes that the user is currently interacting with, and adjusting their resources to ensure satisfactory reponse times.

By dynamically adjusting resources, the daemon seeks to maximize user satisfaction.

The second role of the daemon comes from our desire for a better working environment. In current systems, whenever we run a process that needs more memory than the workstation has (this is not always avoidable), the systems becomes sluggish. A ten-second think time easily leads to a couple of seconds delay for the next mouse movement or keystrokes. This is often annoying enough that we search for some other victim machine to run the job, or stop using the machine while the job is running.

The reason for sluggish responses is that operating systems do not take into account the "importance" of processes in allocating memory. Most of them use a global least-recently-used replacement algorithm or its approximations. If the user stops interacting with a process for a few seconds while a background memory "eater" is running, the process will gradually lose its memory resident pages. When the process is woken up at the next keystroke, it has to page in those pages in order to process the keystroke. Thus, a few seconds of delay is observed by the user.

With kernel mechanisms to adjust the memory allocation, the policy daemon has the opportunity to improve the interactive processes' response time. We identify the following applications which are responsible for most of user inputs, and the typical user interactions:

- window system processes: X server, window manager (e.g. *fvwm*), terminal emulator (*xterm*). Typical user interactions include mouse movements, switching among windows, moving and resizing windows, as well as menu pull-downs.

- command interpreters: various flavors of shells. Typically invoked functions include file name completion, traversing history list, and common commands such as *ls*.

- screen editors: *emacs* and *vi*. Typical interactions are inserting and deleting text, starting a new line, searching and other menu options.

- web browser: *netscape*. Typical interactions include using an internal link, using `BACK` and searching.

The list is particular to the UNIX environment we are familiar with, but similar lists can be easily specified for other environments.

We performed the following experiments to understand what resources the program needs to process the user inputs. Our machine is a 133MHz Pentium PC running Linux and XF86, with 32MB of physical memory and a 1.6GB SCSI disk. We first start the application, and after making sure it is mostly memory resident, invoke various interactions, and measure the screen response time. The time is mostly due to CPU time needed to process the interaction. We then swap out all of the application's memory pages, and invoke the same interactions again. Table 1 shows differences between the response times and the number and types of pages the application has to bring into memory. Since we used *xmon* to measure the response time, the times for X window system processes are not available.

How would the response times be different if some, but not all, pages are in memory? To answer this we use an *EXPECT* script to measure the latency of interactions with *vi* and *emacs* in text mode, that is with no X server present. The script ran the program being tested, and then used a memory intensive program to wip it completely from memory. It then measured the latency of the next interaction with the program. The process repeated with various amounts of *vi* or *emacs* kept resident. The results for emacs are shown in figure 1. The results for vi are shown in figure 2. As can be seen, emacs needs about 160 pages (640K) to respond to the user promptly, while vi needs about 49 pages (196K).

We conducted similar experiments with other applications, and found that they all need some specfic amounts of memory to respond to user input promptly. Table 2 shows the parameters we use for various applications. The numbers are obtained by first swapping the process out of memory, then giving it various user inputs, and observing the resident sizes of

Application	Interaction	Resident Response(sec)	Non-Resident Response(sec)	Total Pages Needed	Text	Library	Data
X server	send letter 'i' to emacs	-	-	49	18	31	0
	send 'i' `\b` `\r` to emacs	-	-	77	33	41	3
fvwm	changing windows	-	-	64	37	18	9
xterm	display+rollback	-	-	114	69	29	16
emacs (in X)	type 'i'	.002	1.2	154	94	13	47
	type 'i' `\b` `\n`	.005	1.5	139	133	87	18
vi (in xterm)	typing 'i'	.003	.794	39	20	15	5

Table 1: Number of pages accessed by various applications to process common user interactions.

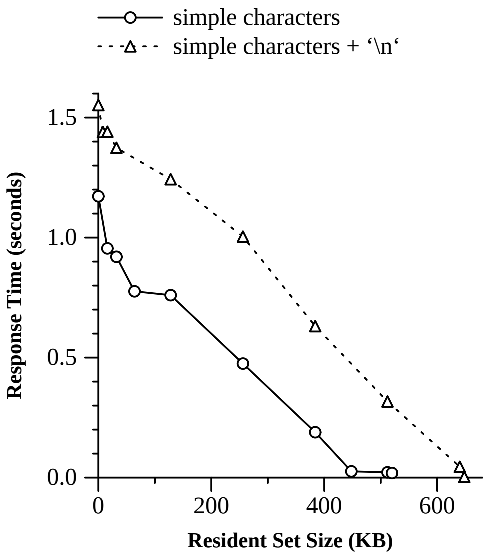

Figure 1: Effect of reserving memory on Emacs' response time to user keystrokes. Two type of interactions are measured: the solid curve shows the latency when the user simply types letters, and the dashed curve shows when the user types a letter, then backspace, and then return. The numbers are gathered using expect scripts, and are average of three tests, with variances all within 20%.

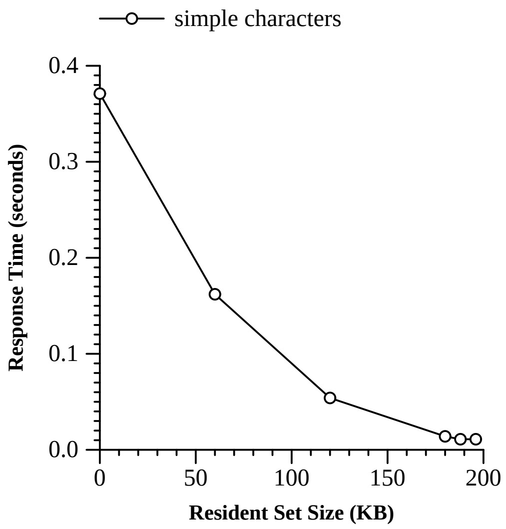

Figure 2: Effecting of reserving memory on vi's response time to user keystrokes. Only one type of interaction, user typing letters, is measured here. The numbers are gathered using expect scripts, and are average of three tests, with variances all within 20%.

Application	Lower Memory Limit
X servers	1MB
xterm	524KB
fvwm	380KB
tcsh	384KB
bash	480KB
emacs (basic typing)	1080KB
emacs (several commands)	2MB
vi	216KB
netscape	3MB

Table 2: Approximate amounts of memory needed to be reserved for interactive applications.

the process in "top". These data, along with the path name of the executable of each application, are specified in a configuration file. The policy daemon reads the file to recognize the interactive processes and sets the corresponding lower memory limits. The daemon also sets the I/O priority of the processes to 1.

Finally, the daemon needs to recognize which of the nominally interactive processes are actually active. The daemon makes the decision by watching the stdin, tty and socket activities for each interactive process. If the process has not shown any activity for a specified period of time, the daemon resets its lower memory limit to zero and lowers its I/O priority. The "time-out" period is adjustable, with the default value being 30 minutes.

We have implemented the kernel mechanisms, the authorization policy and the policy daemon for our PC, which runs Linux. Below we describe the implementations in more detail.

3 Implementation

There are two parts in our implementation, kernel modifications for the new system calls, and the policy daemon.

3.1 Kernel Modifications

Section 2.1 introduced the system calls: `min_memory`, `max_memory`, and `io_priority`. Implementing the calls requires modifying the page stealing algorithm in the VM system, and the elevator algorithm in disk I/O scheduling.

3.1.1 Implementing Memory Limits

In the unmodified Linux kernel, whenever a process running in the kernel needs a new physical page, it first attempts to tear a page from the free page list [17]. If the free page list is empty, it tries to get a free page in one of three ways: it can shrink any memory maps (that is it will traverse the page tables describing any file to memory memory mappings), it can steal a page from the file buffers, or it can steal a page from a process. Successive requests for a free page cycle through the three options. In enforcing upper and lower limits on the physical memory allocated to a process, we focus on the last source of free pages.

When the unmodified kernel tries to steal a page from a process, it determines how many pages to steal from that process before moving on to the next process, taking into account the number of pages resident. Each time a process is visited, the algorithm tries to steal a fixed proportion of the processes resident set.

Our modifications are the following. If a process has a lower limit set on its physical memory, and it is already at its lower limit, the process is skipped when some process tries to steal a page from it. The stealing process simply moves on to another process, the buffers or the memory maps to continue the search.

When a process needs a free page, it is first checked to see whether there is an upper limit on its physical memory, and if so, whether it is at or above its upper limit. If it is, then the process does not get a page from the free page list, the buffers, the memory maps, or other processes. Instead it steals a page from itself.

The kernel also checks the upper limit when it examines a process to decide how many pages to steal from the process. If the process has an upper limit and is above the limit, the kernel takes enough pages away from the process so that it is within the limit. Note that this situation only occurs when an upper limit is first set for a process, or when an exisiting upper limit is lowered.

Whenever both an upper limit and a lower limit are set for a process, they are checked to ensure that the upper limit is higher than the lower limit. In addition, whenever a lower limit is requested, all of the lower limits are checked to ensure that memory is not overcommitted. If a user attempts a request that would violate either of these conditions, it is ignored and an error is returned.

These changes to the kernel result in little additional overhead. The only effect is that when a process has a lower limit set and is at or below its lower limit, the average search for a free page is slightly lengthened, as the process is skipped in the search.

3.1.2 Implementing I/O Priorities

The unmodified Linux kernel uses an elevator algorithm for disk I/O scheduling. The algorithm sorts I/O requests by disk addresses. That is, whenever a process issues a block I/O request, the request is inserted to a device specific queue according to its disk address. The scheduling algorithm attempts to maximize the throughput from disks, but makes no distinction about the relative importance of I/O turnaround time for different processes.

We changed the elevator algorithm to take I/O priorities into account. Disk I/O requests are now sorted first by priority, and then within each priority, by disk addresses. To avoid starvation, we maintain a count of the number of times any request is skipped over by a higher priority request. After the count excedes `skip_threshold`, the request is moved to the next higher priority class. Currently `skip_threshold` is set at 5.

The implementation adds additional CPU overhead because of the slightly more complicated ordering scheme. Overall disk throughput may be reduced since the strict elevator scheme is broken up into priority classes. However, the concern is secondary to the primary concern of improved response time.

3.1.3 Providing Activity Information

In order for the user level daemon to make decisions regarding the allocation of resources, it must be able to determine which nominally interactive processes are actually active and which are being ignored by the user. The information, however, cannot be easily infered from process idle time, since some processes periodically receive messages from the window system as commonly occurs when a process is sent re-display messages. In addition, there are a wide variety of ways that processes can obtain their inputs. Thus, we instrument the kernel to gather statistics on a variety of apparent activity that might qualify as actual user interaction.

For each process, we keep track of reads from stdin, reads from sockets, and reads from ttys. The number of page faults are also remembered. All these statistics are provided to the daemon as a virtual file in the `proc` file system, `/proc/pid/activity`. We also introduce another virtual file, `/proc/pid/mem_limits`, which shows the current values of any upper and lower limits on the physical memory dedicated to process `pid`, any I/O priority, the number of free pages the process has used, and the number of pages stolen from other processes and from the process itself. Together with `/proc/pid/stat` and `/proc/pid/statm`, these files provide a wealth of information about a process, including the page faults statistics, the memory resident set and the current status of the process.

3.2 User Level Policy Daemon

The policy daemon depends on a FIFO to be notified of user requests, and the configuration files to handle default settings for interactive processes. It is started at boot time along with the other standard daemons. We use the `register_resource_daemon` system call to notify the system of the pid of the daemon. The kernel remembers the pid of the daemon to check upon invocations of the system calls. A daemon can only be registered if none has yet been registered, or if the previous daemon has exited. It can only be registered by root.

3.2.1 Handling User Requests

We developed library routines that pass user requests to the daemon process. The routine writes a message containing the user id of the requester to the daemon's FIFO and writes a confirmation to a special file owned by the owner of the process. We have placed these files in `/var/adm/resources`, one file for each user. The daemon polls the FIFO and, upon seeing a new message claiming to be from a user, checks that user's confirmation file. Since each user can only write to his or her own file, the daemon is sure about the identity of the user making the request. The daemon can then go through the authorization checks to see if the user can make the request. If so, a system call is issued to the kernel. The kernel then checks the pid of the process issuing the system call to ensure that it is the previously registered daemon.

We also developed several programs to allow a user to set the memory limits and I/O priorities through the command line. Chief among these are:

- `launchapp priorities` *program_name params*: launches an application with priorities determined by `priorities` and with command line parameters `params` to the program;
- `adjustapp` *pid* `priorities` : adjusts the priorities of *pid* acording to `priorities`.

The options for `priorities` are:

- `-u####`: sets an upper limit of `####` on the physical memory devoted to the process;
- `-l####`: sets a lower limit of `####` on the physical memory devoted to the process;
- `-p[inb]`: sets the I/O priority of the process to interactive(1), normal(0) or background(-1). Only one can be chosen.

A key point that makes these commands worthwhile is that they can be embedded in scripts or window manager "dots" files. Also, many applications can have default values set in the daemon's configuration file. Thus in typical system use the user will not notice any burdensome requirements when using the new resources.

3.2.2 Dynamic Resource Adjustments

The second job of the daemon process is to implement a "user-oriented" resource policy, which separates the processes into interactive ones and non-interactive ones, and reserves resources for interactive processes to ensure user satisfaction.

The configuration file described in Section 2.3 lists the names of important interactive applications and their resource needs. Periodically (every 3 seconds in our environment), the daemon wakes up, and checks the new processes that were created since the last time it went to sleep. It does this by scanning the `proc` directory for any process files with pid's larger than the largest present when the daemon last checked. If any of the new processes has an executable that is listed in the configuration file, the daemon adds the process to it's list of those to be monitored and sets it's memory limits and I/O priority as specified in the configuration file.

Less often (every 30 seconds in our environment) the daemon wakes up and checks the activity of all processes which have lower limits on their physical memory. This activity information can be obtained from the various files detailed in 3.1.3. The daemon considers a process to be have been active during the last wakeup cycle if the process has shown stdin, tty or socket activities, or if the process has had page faults. If the daemon discovers that no activities are detected for a time-out period (30 minutes in our environment), the daemon decides that the user has stopped using the process. The daemon then resets the process's lower memory limit to 0 (i.e. no lower limit), thus allowing it to be paged out of memory. At any time, however, if some activities are detected, the original lower limit is restored.

We have implemented this policy and it appears to perform well. However, the policy is by no means optimal. On the other hand, since it is implemented at user level, it can be changed easily without involving the kernel. We plan to experiment with more policies on dynamically adjusting resource allocations. On our current test system, the daemon consumes approximately 1/10 of 1% of CPU time while monitoring the activity of more than a dozen interactive processes.

4 Performance

A natural question about our resource adjustment policy is how effective it is, and what impacts it has on non-interactive processes. In this section we attempt to answer the questions with experimental results. All tests in this section were done on a 133MHz Pentium, with 16MB of memory and 2 1.2GB disk drives.

4.1 Maintaining performance of interactive applications, measured with *EXPECT* scripts

In order to measure the effects of `min_memory` on interactive processes, we use *EXPECT* to mimic simple user interactions with *emacs*, *vi*, and *bash*. Each test involves starting up the interactive process with various amounts of memory reserved, and then starting a memory intensive process to compete with the interactive one. The *EXPECT* script then begin various typing interactions and measures the latency of each interaction. The script pauses for various periods of time between each interaction, ranging from no time to 30 seconds. Since these tests are intended to measure delays during ongoing work, the script performs each interaction several times to allow the pages required to support the interaction to be paged into memory for the first time or to be created as necessary. After these startup interactions, the script then begins to measure interactions' latencies.

For these tests we keep the *EXPECT* testing program fully resident in memory so that our timing does not include any page faults from it. For each each combination of test, background process and pause interval we report the minimum, maximum and average latency of the interaction.

We ran the tests with the following interactions:

- *bash* : *cat* a file in the *bash* shell (Table 3 and Table 4),
- *emacs1* : typing characters in *emacs* in text mode, no X server (Table 5 and Table 6),
- *emacs2* : typing characters and `\r` in *emacs* with no X server present (Table 7 and Table 8),
- *vi* : typing characters in *vi* (Table 10 and Table 9).

We tested each of these in the presence of two numerical computations:

- *tridiag* : a tridiagonal matrix computation (process size 70MB); thrashes badly,
- *es* : an electrostatic computation (process size 100MB); alternates between being CPU intensive and thrashing.

pages	*Interval between interactions*														
	0 sec.			1 sec.			5 sec.			10 sec.			30 sec.		
0	16	**16**	16	194	**204**	214	391	**490**	647	1849	**1881**	1919	601	**2127**	4425
35	16	**16**	16	204	**214**	234	851	**881**	905	1726	**1735**	1752	1522	**2357**	2855
70	16	**16**	16	194	**194**	194	151	**434**	946	751	**891**	1021	1098	**1311**	1594
105	15	**16**	16	194	**207**	224	119	**287**	377	421	**440**	474	397	**729**	923
122	16	**16**	16	184	**191**	194	357	**407**	437	403	**447**	504	464	**578**	755

Table 3: Running `cat` with a small file under `bash` with `es` in background. Latencies (min/**avg**/max) in ms.

pages	*Interval between interactions*														
	0 sec.			1 sec.			5 sec.			10 sec.			30 sec.		
0	16	**20**	39	36	**63**	85	40	**119**	313	173	**447**	1053	1116	**1665**	1938
35	16	**17**	18	34	**48**	59	55	**100**	140	179	**423**	724	1452	**1617**	1855
70	16	**18**	21	36	**98**	298	94	**159**	283	187	**416**	762	809	**1104**	1543
105	17	**19**	24	31	**53**	78	52	**107**	218	71	**298**	508	349	**426**	591
122	16	**17**	22	38	**55**	77	31	**50**	64	16	**137**	464	261	**306**	412

Table 4: Running `cat` with a small file under `bash` with `tridiag` in background. Latencies (min, **avg**, max) in ms.

pages	*Interval between interactions*														
	0 sec.			1 sec.			5 sec.			10 sec.			30 sec.		
0	1	**1**	1	1	**2**	2	2	**2**	2	1052	**1173**	1388	1021	**1554**	2552
40	1	**1**	1	2	**2**	2	2	**2**	2	322	**807**	1055	820	**1036**	1315
80	1	**1**	1	2	**2**	2	2	**2**	3	97	**131**	151	216	**913**	1695
120	1	**1**	1	2	**2**	2	2	**253**	738	2	**2**	2	556	**600**	666
140	1	**1**	1	2	**2**	2	2	**3**	4	2	**10**	23	1	**16**	47
160	1	**1**	1	2	**3**	4	2	**9**	20	4	**6**	11	1	**3**	5

Table 5: Typing characters in `emacs` with `es` in background. Latencies (min/**avg**/max) in ms.

pages	*Interval between interactions*														
	0 sec.			1 sec.			5 sec.			10 sec.			30 sec.		
0	1	**2**	7	2	**11**	44	2	**175**	629	55	**276**	1806	744	**1225**	1644
40	1	**3**	10	2	**9**	35	2	**24**	76	132	**314**	591	676	**873**	1164
80	1	**4**	17	2	**10**	28	15	**58**	172	106	**166**	234	108	**487**	1339
120	1	**4**	16	2	**10**	32	2	**10**	43	2	**15**	35	2	**7**	25
140	1	**3**	12	3	**7**	20	2	**4**	14	1	**5**	19	1	**15**	42
160	1	**5**	17	2	**12**	47	2	**7**	22	2	**6**	24	1	**6**	24

Table 6: Typing characters in `emacs` with `tridiag` in background. Latencies (min/**avg**/max) in ms.

pages	*Interval between interactions*														
	0 sec.			1 sec.			5 sec.			10 sec.			30 sec.		
0	3	**3**	3	3	**3**	3	3	**25**	69	687	**1220**	1625	662	**2695**	4441
57	3	**3**	3	3	**4**	3	3	**8**	19	1406	**1472**	1531	1569	**2330**	3072
114	3	**3**	3	3	**3**	3	3	**4**	4	621	**735**	905	860	**1001**	1117
171	3	**3**	3	3	**3**	3	3	**3**	3	3	**3**	3	4	**4**	4

Table 7: Typing `'i' + 'r'` in `emacs` with `es` in background. Latencies (min/**avg**/max) in ms

pages	*Interval between interactions*														
	0 sec.			1 sec.			5 sec.			10 sec.			30 sec.		
0	3	**4**	6	6	**6**	7	3	**64**	274	42	**248**	470	1224	**1597**	2090
57	2	**4**	6	3	**7**	15	3	**43**	172	168	**294**	380	1100	**1371**	1605
114	3	**4**	5	4	**6**	7	3	**3**	3	144	**294**	540	480	**606**	715
171	3	**3**	5	4	**5**	7	3	**4**	4	3	**4**	6	51	**214**	281

Table 8: Typing `'i' + 'r'` in `emacs` with `emacs` in background. Latencies (min/**avg**/max) in ms.

pages	*Interval between interactions*														
	0 sec.			1 sec.			5 sec.			10 sec.			30 sec.		
0	1	**1**	2	1	**153**	493	1	**19**	88	62	**234**	483	500	**598**	813
17	1	**3**	12	4	**114**	199	1	**9**	28	66	**201**	274	228	**422**	716
34	1	**1**	1	3	**234**	83	1	**46**	129	38	**111**	184	31	**79**	116
51	1	**1**	3	1	**3**	4	1	**1**	1	1	**1**	1	1	**11**	37

Table 9: Typing a single character in `vi` with *tridiag* in background. Latencies (min/**avg**/max) in ms.

pages	*Interval between interactions*														
	0 sec.			1 sec.			5 sec.			10 sec.			30 sec.		
0	1	**1**	1	1	**1**	1	415	**462**	486	577	**668**	775	587	**766**	986
17	1	**1**	1	1	**1**	1	30	**366**	609	436	**516**	656	562	**686**	760
34	1	**1**	1	1	**1**	1	21	**131**	211	90	**124**	181	229	**329**	519
51	1	**1**	1	1	**1**	1	1	**1**	1	1	**1**	1	1	**1**	1

Table 10: Typing a single character in `vi` with *tridiag* in background. Latencies (min/**avg**/max) in ms.

As can be seen from the charts, as more memory was allocated to the interactive process, latencies were generally reduced to times that are imperceptible to a human user. Since the *bash* test requires file access and creating a new process with a new address space, it does not decrease as significantly as the others. Since these are tests with real background applications, there is significant variation in some of the numbers, in particular the major outlier in Table /refemacs-results-1-es.

4.2 Maintaining performance of interactive applications, measured with xmon

.

To measure the latency of various real-world interaction with *emacs* and *netscape* under X, we used *xmon* to timestamp each message between the X server and the application. We then measured the latency of various interactions, while running a thrashing application in the background. We measured the latencies with 1/2, 2/3, and 1 times the physical memory required by the process resident. The thrashing application was an extreme worst case process which cycles through a large array, writing to a page and then moving on to the next page.

For *emacs* we measured the latency of displaying the window when switching from another virtual desktop, of moving the mouse into the window, of pressing the meta-key, of entering the command research-forward, and of the actual search and resulting redisplay. Table 11 shows that if a full 400 pages are devoted to *emacs*, overall latency for this simple sequence of operations can be significantly decreased.

For *netscape* we browsed a local file, in this case a copy of the *Linux Documentation Project* web page. We measured the latency of displaying the window, of the mouse entering the window, of clicking on a link internal to the file, then clicking `BACK` and then repeating a previous search for *Linus*. In Table 12 we see that allocating 692 pages (2768 KB) would dramatically decrease overall latency for these operations.

Since *xmon* is only able to measure the time between the message from X and the response from the application, we are not able to quantify the latency as X itself pages into memory.

4.3 Effects on background processes

In order to measure the effects of lower limits on some applications on other, memory intensive applications, we ran the two memory-intensive applications above in the presence of an artificial application that simply occupied memory. The lower limits for this program were set in a range from 0KB to 8MB. As can be seen in Table 13, as the amount of memory reserved for the memory resident process is increased, the turnaround time increased by less than 20%, until 8MB were reserved, at which time a majority of the memory was committed to other things: kernel, free page list, memory resident process, *EXPECT* test program).

5 Related Work

There are many early studies on resource management, especially memory management [15, 3, 18, 2, 1, 24, 9, 10, 7]. Except for some early multiprogramming systems [15], most operating systems adopted the kernel-based global management of memory [3, 18, 2, 1, 24]. Hence, most of the research focused on finding algorithms that would optimize the throughput of the system. Denning's working set paper[10] gives a very good survey on the research results up to the 80's, and shows that the working-set principle is an effective technique to control multiprogramming level and maximize throughput. Unfortunately, true working-set policy is very hard to implement. Instead, most systems use a FIFO with second chance or CLOCK algorithm [18, 2, 1].

Recently the operating system community has been looking into the issues in incorporating application control on resource management in modern kernels [16, 5, 6, 4, 11]. A number of research projects studied the policy and implementation issues of giving programmers the oppurtunity to manage resources given to an application [16, 5, 6], and better kernel structures to facilitate application control [4, 11]. However, the focus in these studies is on application control, not user control. The systems still consider the global resource allocation to be the job of the kernel, and do not investigate issues of giving users authority to manage resources.

There are also proposals to use new schemes to manage global resource allocation and scheduling [16, 23, 12]. For example, the money-market approach [16] allocates resources as if they were commodities sold in a market, and applications are required to "buy" their resources; the lottery scheduling approach [23] uses a probablistic scheme to allocate resources so that propotional share can be achieved without starvation. These approaches still keep the kernel as the only one who has authorities on resource management, and treat interactive processes and non-interactive one in the same way.

Modern UNIX systems provide a system call, "mpin" or "mlock", which allows super user to pin part of the address space of certain application in memory [13]. However, using "mpin" for our purpose would require

	Number of emacs pages resident			
emacs interaction	0	187	374	400
display	3231	1283	576	517
mouse enters window	300	237	230	115
pressing meta key	1580	556	77	65
typing first key in meta-window	89	39	4	3
M-x re-search-forward	425	153	151	126
actual search and redisplay	207	153	151	126
Total latency:	4270	2599	1057	831

Table 11: Latency of interactions with emacs in X, in the presence of a thrashing background process.

	Number of Netscape pages resident			
Netscape interaction	0	346	462	692
display	3231	2311	1448	385
move mouse into window	210	165	146	17
click on internal link	3840	2263	2410	451
... then click on "BACK"	1059	1221	1147	983
Do a repeat of a previos search.	294	152	169	159
Total latency:	8634	6112	5320	2395

Table 12: Latency of interactions with emacs in Netscape, in the presence of a thrashing background process.

application	*KB kept resident (unavailable to these applications)*						
	0 KB	512 KB	1024 KB	1536 KB	2048 KB	4096 KB	8192 KB
`tridiag`	578	569	567	574	506	508	619
`es`	3351	3326	3369	3646	3496	3970	15249

Table 13: Effect of keeping other processes resident on test memory intensive applications. Turnaround times in seconds.

that the user runs in superuser mode, the application be changed to pin its relevant address space in memory, and the pinned pages will not be paged out, even if the process is stopped. It also do not address the I/O priority problem. Thus, “mpin” would not be the appropriate mechanism to use for user level resource management.

Our dynamic resource adjustment approach borrows ideas from soft real-time systems [21, 22, 19, 20]. In some sense, we are viewing a PC as a soft real-time system: interactive processes have deadlines in the sense that they must respond to user inputs within a human tolerable threshold. Similar to real time systems, our goal is to maximize throughput under the constraint of the interactive jobs meeting their deadlines most of the time. Most real time systems, however, are not general purpose computing systems. Studies in this area often assume precise knowledge on each job's resource requirements and computation time, which is often unavailable for general purpose computing environments. To our knowledge, our characterization of the memory needs of interactive applications in section 2.3 is the first effort on characterizing the performances of interactive processes using real-time system techniques.

Finally, some operating systems like Windows NT [8] set memory and CPU time limits on processes. Unfortuantely, the systems do not guarantee the minimum memory required by interactive applications. Thus, multiple background processes can still collectively thrash the machine, making interactive processes very slow. Neither do the systems address the I/O priorities. On the other hand, although we focus most of our discussion on UNIX-like systems, user-oriented resource management applies to other multi-programming operating systems.

6 Conclusion and Future Work

As personal computers become more common in various organizations, it is time to reconsider the assumptions made by traditional operating systems on resource management. In this paper we propose user-oriented resource scheduling, which allows authorized users to influence the resource allocation decisions made by the kernel. The goal of the approach is to give user greater control over the computer, and to maximize user satisfaction in daily use of the computer.

User-oriented resource management includes three components: new system calls to allow users to specify memory limits and I/O priorities, authorization policies that decide which user can make what request on resource allocation, and a user-level policy daemon that implements the authorization policy and makes the system calls on users' behalf. Since interaction applications have greater impact on user's perceived usability of the system, the policy daemon also employs a dynamic resource adjustment policy that reserves resources for interactive applications to ensure prompt responses to user inputs.

Our implementation of the approach in Linux demonstrated that user-oriented resource management can be easily incorporated into existing kernels. Our experiments also show that common interactive applications need only modest amount of memory in order to process user interactions promptly, and that reserving the memory for those applications significantly improve the usability of the machine with only slight slowdown of non-interactive applications.

There are many limitations in our work. Our current model of the authorization policy is designed for personal computers; other environments may need a more expressive framework. Current schemes on identifying active interactive applications need to be more precise. We have not fully studied the performance implications of I/O priorities (we plan to experiment more on this issue in the coming weeks). *(more limitations?...)*

We plan to extend our work in a number of directions. We will look into how the resource reservation scheme can be extended to support multi-media applications. We will also investigate how user-oriented resource scheduling facilitate sharing of people's workstations. In addition, putting limits on resources help thwart “denial of service attacks” from malicious parties and we plan to look more into that. We are also interested in extending our framework to provide better working environment on time-sharing systems. Finally, we plan to look into the broader question of “quality of service” in an operating system.

References

[1] Ozalp Babaoglu and William Joy. Converting a swap-based system to do paging in an architecture lacking page-reference bits. In *Proceedings of the 8th SOSP, Operating Systems Review 15(5)*, pages 78–86, December 1981.

[2] Maurice J. Bach. *The Design of the UNIX Operating System.* Prentice-Hall Software Series, 1986.

[3] A. Bensoussan, C.T. Clingen, and R.C. Daley. The multics virtual memory: Concepts and design. In *Communications of the ACM, 15(5)*, pages 308–318, May 1972.

[4] Brian N. Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gun Sirer, Marc E. Fiuczynski, David Becker, Craig Chambers, and Susan Eggers. Extensibility, safety and performance in the SPIN operating

system. In *Proceedings of the 15th SOSP*, pages 267–284, December 1995.

[5] Pei Cao, Edward W. Felten, and Kai Li. Application-controlled file caching policies. In *Proc. USENIX Summer 1994 Technical Conference*, pages 171–182, June 1994.

[6] Pei Cao, Edward W. Felten, and Kai Li. Implementation and performance of application-controlled file caching. In *Proc. First USENIX Symposium on Operating Systems Design and Implementation*, pages 165–178, November 1994.

[7] R. Carr and J. Hennessy. WSCLOCK-a simple and effective algorithm for virtual memory management. In *Proceedings of the 8th SOSP, Operating Systems Review 15(5)*, pages 87–95, December 1981.

[8] Helen Custer. *Inside Windows NT*. Microsoft Press, 1992.

[9] Peter J. Denning. The working set model for program behavior. *Communications of the ACM*, 11(5):323–333, May 1968.

[10] Peter J. Denning. Working sets past and present. *IEEE Transactions on Software Engineering*, SE-6(1):64–84, January 1980.

[11] D. R. Engler, M. F. Kaashoek, and J. O'Toole. ExoKernel: An operating system architecture for application-level resource management. In *Proceedings of the 15th SOSP*, pages 251–266, December 1995.

[12] Gregory R. Ganger and Yale N. Patt. The process-flow model: Examining i/o performance from the system's point of view. In *Proceedings of the ACM Sigmetrics Conference*, pages 86–97, May 1993.

[13] R. Gingell, J. Moran, and W. Shannon. Virtual memory architecture in SunOS. In *Proceedings of the USENIX Conference*, Summer 1987.

[14] Greg Hankins. *Linux Documentation Project Home Page*. http://sunsite.unc.edu/mdw/linux.html, 1996.

[15] Per Brinch Hansen. The nucleus of a multiprogramming system. In *Communications of the ACM, 13(4)*, pages 238–250, April 1970.

[16] Kieran Harty and David R. Cheriton. Application-controlled physical memory using external page-cache management. In *The Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 187–197, October 1992.

[17] Michael K. Johnson. *The Kernel Hacker's Guide 0.7*. http://www.redhat.com:8080/HyperNews/get/khg.html, 1996.

[18] Henry M. Levy and Peter Lipman. Virtual memory management in vax/vms. In *Computer, 15(3)*, pages 35–41, March 1982.

[19] Cliff Mercer, Ragunathan Rajkumar, and Jim Zelenka. Temporal protection in real-time operating systems. In *Proceedings of the 11th IEEE Workshop on Real-Time Operating Systems and Software*, pages 79–83, May 1994.

[20] Clifford W. Mercer, Stefan Savage, and Hideyuki Tokuda. Processor capacity reserves: Operating system support for multimedia applications. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, May 1994.

[21] J. A. Stankovic and K. Ramamritham. The spring kernel: A new paradigm for real-time systems. In *IEEE Software*, pages 62–72, May 1991.

[22] Hideyuki Tokuda, Tatsuo Nakajima, and Prithvi Rao. Real-time mach: Towards a predictable real-time system. In *Proceedings of USENIX Mach Workshop*, October 1990.

[23] Carl A. Waldspurger and William E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *Proceedings First Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–12. USENIX, November 1994.

[24] Michael Young, Avadis Tevanian, Richard Rashid, David Golub, Jeffrey Eppinger, Jonathan Chew, William Bolosky, David Black, and Robert Baron. The duality of memory and communication in the implementation of a multiprocessor operating system. In *Proceedings of the Eleventh Symposium on Operating Systems Principles*, pages 63–76, November 1987.