

EXTENSION OF THE getusage() SYSTEM CALL TO PROVIDE INFORMATION ON CONTEXT SWITCHES

A REPORT ON THE PACKAGE SUBMITTED BY

N KRISHNA

18PT23

K R CHARAN

18PT41

SUBJECT: OPERATING SYSTEMS – 18XT44



DEPARTMENT OF APPLIED MATHEMATICS AND
COMPUTATIONAL SCIENCES

PSG COLLEGE OF TECHNOLOGY

COIMBATORE - 641 004

CONTENTS

1. Introduction	3
2. Description	4
3. System Calls Used	6
4. Tools and Technologies	7
5. Work Flow	8
6. Results and Discussion	15
7. Conclusion	17
8. Bibliography	18

Introduction

A context switch occurs when the CPU switches from one process or a thread to a different process or thread. Context switching allows for one single processor to handle numerous processes or threads without the need for additional processors. Any operating system that allows for multitasking heavily relies on the use of this feature of context switching to allow different processes to run at the same time. This activity is important for several reasons. A program that monopolizes the processor lowers the rate of context switches because it does not allow much processor time for the other processes' threads. A high rate of context switching means that the processor is being shared repeatedly—for example, by many threads of equal priority. A high context-switch rate often indicates that there are too many threads competing for the processors on the system.

Context switches occur in the process scheduler, which is at the core of the operating system. Context switches can be classified into two categories:

1. Voluntary Context Switches – These kind of context switches occur when the process or thread yields control to the CPU because of waiting for an I/O, sleeping etc.
2. Involuntary Context Switches – These kind of context switches occur when the process or thread is interrupted by the scheduler, which may occur when the timeslice of the process expires or a higher priority process arrives, etc.

Description

The `getrusage()` system call returns resource information about a given process. The `rusage` structure it uses has a number of fields, not all of which are necessarily filled in by a given operating system. In this project, the Linux kernel code has been examined to see what fields it fills in for the `rusage` structure and extend the implementation so that the `getrusage()` system call returns information about context switches.

For this package, we initially take a look at the `getrusage()` system call.

Recording Context Switches

Once primary changes to the structure are made, then the corresponding kernel code is to be modified accordingly. Context switches occur in the scheduler. The scheduler is a kernel function called `schedule()`, that gets called from other system call functions (usually when a process goes to sleep waiting for I/O), after every system call and after some interrupts. When invoked, the scheduler:

1. Performs some basic periodic tasks, like handling interrupt service routines (not a concern of this project)
2. Chooses one process to execute according to the scheduling policy
3. Dispatches the chosen process to run

The Linux scheduler contains different built-in scheduling strategies with `SCHED_OTHER` as the default.

Implementation program

In the implementation program, we will simulate access control to a communal bathroom used by both sexes. Individuals will be simulated by threads, and the access controls must be implemented by data structures and synchronization primitives to cause individuals to wait or proceed.

Access to the communal bathroom is controlled by a sign on the door indicating that the bathroom is in one of three states: –

- a. Vacant
- b. Occupied by men
- c. Occupied by women

There is no valid state in which the bathroom is occupied by both women and men. When a person approaches the bathroom, he or she may enter if and only if it is either vacant or occupied by others of the same sex. Otherwise, he or she must wait until it becomes vacant. Upon entering, the user sets the sign on the door to the appropriate state if it was previously vacant.

Multiple people may be waiting at the same time. Obviously, waiting people will all be of the same sex, which is the opposite of those in the bathroom. Arriving people may “jump the queue” if they are of the same sex as those using the bathroom. When the last person leaves the bathroom, he/she sets the sign to indicate that it is vacant. This allows all waiting people to enter one at a time, setting the sign appropriately.

System Calls Used

1. The primary system call that is used in this project is `getrusage()`, whose functionality we intend to extend.

The `getrusage()` function provides measures of the resources used by the current process or its terminated and waited-for child processes. If the value of the `who` argument is `RUSAGE_SELF`, information shall be returned about resources used by the current process. If the value of the `who` argument is `RUSAGE_CHILDREN`, information shall be returned about resources used by the terminated and waited-for children of the current process.

Other system calls used in the implementation program are as follows:

2. `exit()`- This function terminates the process normally. Its status value is returned to the parent process. Generally, a status value of 0 or `EXIT_SUCCESS` indicates success, and any other value or the constant `EXIT_FAILURE` is used to indicate an error. The function performs the following operations:
 - a. Flushes unwritten buffered data.
 - b. Closes all open files.
 - c. Removes temporary files.
 - d. Returns an integer exit status to the operating system.

Tools and Technologies

C

The C programming language is a computer programming language that was developed to do system programming for the operating system UNIX and is an imperative programming language. C was developed in the early 1970s by Ken Thompson and Dennis Ritchie at Bell Labs. It is a procedural language, which means that people can write their programs as a series of step-by-step instructions. C is a compiled language.

Because the ideas behind C are kept close to the design of the computer, the compiler (program builder) can generate machine code/native code for the computer. Programs built in machine code are very fast. This makes C a good language for writing operating systems. Many operating systems, including Linux and UNIX, are programmed using this language. The language itself has very few keywords, and most things are done using libraries, which are collections of code for them to be reused.

C is available for many different types of computers. This is why C is called a "portable" language. A program that is written in C and that respects certain limitations can be compiled for many different platforms.

Ubuntu (Operating System)

Ubuntu is a free and open source Linux Distribution. It incorporates all the features of a Unix OS with an added customizable GUI, which makes it popular in universities and research organizations. Ubuntu is primarily designed to be used on personal computers, although server editions does also exist.

Ubuntu is built on Debian's architecture and infrastructure, and comprises Linux server, desktop and discontinued phone and tablet operating system versions. Ubuntu packages are based on packages from Debian's unstable branch, which are synchronized every six months. Both distributions use Debian's deb package format and package management tools. Debian and Ubuntu packages are not necessarily binary compatible with each other, however, so packages may need to be rebuilt from source to be used in Ubuntu.

Work Flow

To begin with, the kernel source code is immutable in a given linux system as it is pre-built. Therefore, the required kernel repository that matches the version on the user's system needs to be cloned from the internet using git in the terminal.

The definition of the `getrusage()` system call can be found in `/src/linux/kernel/sys.c`. The system call itself is `void getrusage()` which calls the routine `getrusage()`. It is observable that the filled-in fields of the `rusage` structure come from the structure `task_struct`. This latter structure contains information such as state, signal etc. about each process or thread (task) in the system. Its definition can be found in `/include/linux/sched.h`.

Once we have identified the `task_struct` fields that are used to fill in the `rusage` structure, we should find where these `task_struct` fields are initialized and modified in the source code. It is necessary to look in `kernel/fork.c` and `kernel/exit.c` for initialization and updates of these fields. We can look in `kernel/sys.c` and files in the directory `/usr/src/linux/` to see where the `task_struct` fields are updated.

We will first need to add fields to the `task_struct` to keep track of context switches for each process. Variables are necessary for both parent and children processes. Once we have the structure changes in place, there is a need to modify appropriate code to return the value of these fields for a call to `getrusage()`. The code accumulates values for child processes when these processes exit (as done for other fields in `kernel/exit.c`).



```

struct prev_cputime prev_cputime;
//Initialization of variables to record context switches alongside other resource counters
unsigned long nvcsw; //Voluntary Context Switches
unsigned long nvcsw; //Involuntary Context Switches
unsigned long nvcsw; //Voluntary Context Switches for the child process
unsigned long cnvcsw; //Involuntary Context Switches for child process
unsigned long min_flt, maj_flt, cmin_flt, cmaj_flt;
unsigned long inblock, oublock, cminblock, coublock;
unsigned long naxrss, cxarss;
struct task_accounting toacc;

/*
 * Cumulative ns of schedule CPU time fo dead threads in the
 * group, not including a zombie group leader. (This only differs
 * from jiffies_to_ns((ttime + stime) if sched_clock uses something
 * other than jiffies.)
 */
unsigned long long sum_sched_runtime;

/*
 * We don't bother to synchronize most readers of this at all,
 * because there is no reader checking a limit that actually needs
 * to get both rlim_cur and rlim_max atomically, and either one
 * alone is a single word that can safely be read normally.
 * getrlimit/setrlimit use task_lock(current->group_leader) to
 * protect this instead of the siglock, because they really
 * have no need to disable irq.
 */
struct rlimit rlim[RLIM_NLIMITS];

#ifdef CONFIG_BSD_PROCESS_ACCT
struct pacct_struct pacct; /* per-process accounting information */
#endif
#ifdef CONFIG_TASKSTATS
struct taskstats *stats;
#endif
#ifdef CONFIG_AUDIT
unsigned audit_tty;
struct tty_audit_buf *tty_audit_buf;
#endif

/*
 * Thread is the potential origin of an oom condition; kill first on
 * oom
 */
bool oom_flag_origin; /* OOM kill score adjustment */
short oom_score_adj; /* OOM kill score adjustment min value.
short oom_score_adj_min; /* Only settable by CAP_SYS_RESOURCE. */
struct mm_struct *oom_mm; /* recorded mm when the thread aroun not

```

Figure 3: Definition of variables for both parent and child processes in the definition of struct signal_struct

Linux maintains a counter that is incremented whenever a context switch occurs. This increment occurs in the schedule() function of core.c, when the process identified by the task_struct pointer variable prev switches to the task_struct pointer variable next (where prev and next are different). At this point, we insert a statement to increment the total number of context switches (both voluntary and involuntary) for the process pointed to by prev (since you will keep track of the number of context switches from a process rather than to a process so you should use prev rather than next).

You will then have the total number of context switches, both voluntary and involuntary. At this point, a voluntary context switch means the process is in a state other than TASK RUNNING and is most likely waiting for I/O. TASK RUNNING is used as the state for both running and ready processes. Therefore, if prev->state is not TASK RUNNING then the voluntary context switch count can be incremented as well.

```

/*
 * Make sure that signal_pending_state()->signal_pending() below
 * can't be reordered with __set_current_state(TASK_INTERRUPTIBLE)
 * done by the caller to avoid the race with signal_wake_up().
 */
rq_lock(rq, &rf);
smq_mb__after_spinlock();

/* Promote REQ to ACT */
rq->clock_update_flags <<= 1;
update_rq_clock(rq);

switch_count = &prev->nvcsw; // This process is in the TASK_RUNNING state, hence allowing for recording of voluntary context switches
if (!preempt && prev->state) {
    if (unlikely(signal_pending_state(prev->state, prev))) {
        prev->state = TASK_RUNNING;
    } else {
        deactivate_task(rq, prev, DEQUEUE_SLEEP | DEQUEUE_NOCLOCK);
        prev->on_rq = 0;

        if (prev->in_lowlat) {
            atomic_inc(&rq->nr_lowlat);
            delayacct_blkio_start();
        }
    }
}

/*
 * If a worker went to sleep, notify and ask workqueue
 * whether it wants to wake up a task to maintain
 * concurrency.
 */
if (prev->flags & PF_WQ_WORKER) {
    struct task_struct *to_wakeup;
    to_wakeup = wq_worker_sleeping(prev);
    if (to_wakeup)
        try_to_wake_up_local(to_wakeup, &rf);
}

switch_count = &prev->nvcsw; // This process is no longer in the TASK_RUNNING state, hence allowing for recording of involuntary context switches
// This may also mean that the process is waiting for an I/O process
}

next = pick_next_task(rq, prev, &rf);
clear_task_need_resched(prev);
clear_preempt_need_resched();

if (likely(prev != next)) {
    rq->nr_switches++;
    rq->curr = next;
}

```

Figure 4: Updating the switch_count variable to hold the context switches based on the state of the process in the schedule function

```

/* but we want to avoid the race with thread_group_cputime() which can
 * see the empty ->thread_head list.
 */
task_cputime(tsk, &utime, &stime);
write_seqlock(&sig->stats_lock);
sig->utime += utime;
sig->stime += stime;
sig->gttime += task_gtime(tsk);
sig->nin_flt += tsk->nin_flt;
sig->naj_flt += tsk->naj_flt;
sig->inblock += task_get_inblock(tsk);
sig->outblock += task_get_outblock(tsk);
task_to_accounting_add(&sig->loac, &tsk->loac);
sig->sum_sched_runtime += tsk->se.sum_exec_runtime;
sig->nr_threads--;

// Resource counter to collect the number of voluntary and involuntary context switches when a process is exiting the critical section
sig->nvcsw += tsk->nvcsw;
sig->nivcsw += tsk->nivcsw;
__unhash_process(tsk, group_dead);
write_sequnlock(&sig->stats_lock);

/*
 * Do this under ->siglock, we can race with another thread
 * doing sigqueue_free() if we have SIQQUEUE_PREALLOC signals.
 */
flush_sigqueue(&tsk->pending);
tsk->sigband = NULL;
spin_unlock(&sigband->siglock);

__cleanup_sigband(&sigband);
clear_tsk_thread_flag(tsk, TIF_SIGPENDING);
if (group_dead) {
    flush_sigqueue(&sig->shared_pending);
    tty_kref_put(tty);
}
}

static void delayed_put_task_struct(struct rcu_head *rhp)
{
    struct task_struct *tsk = container_of(rhp, struct task_struct, rcu);

    perf_event_delayed_put(tsk);
    trace_sched_process_free(tsk);
    put_task_struct(tsk);
}

void release_task(struct task_struct *p)
{
    struct task_struct *leader;
}

```

Figure 5: Updating the exit_signal() function to record context switches when a process exits the critical section

```

Open  [icon]  *exit.c  Save  [icon] [icon] [icon]
~/Documents/Kernel copy/Kernel/Kernel

sys.c  *exit.c  *core.c  *sched.h  *resource.h  *signal.h

/* processes cc has previously received. All these
 * accumulate in the parent's signal_struct c++ fields.
 *
 * We don't bother to take a lock here to protect these
 * p->signal fields because the whole thread group is dead
 * and nobody can change them.
 *
 * psig->stats_lock also protects us from our sub-threads
 * which can reap other children at the same time. Until
 * we change k_getrusage()-like users to rely on this lock
 * we have to take ->siglock as well.
 *
 * We use thread_group_cputime_adjusted() to get times for
 * the thread group, which consolidates times for all threads
 * in the group including the group leader.
 */
thread_group_cputime_adjusted(p, &tgttime, &tgtstime);
spin_lock_irq(&current->sigband->siglock);
write_seqlock(&psig->stats_lock);
psig->cutime += tgttime + sig->cutime;
psig->cstime += tgttime + sig->cstime;
psig->cgtime += task_gtime(p) + sig->gtime + sig->cgtime;
psig->cnin_flt +=
    p->nin_flt + sig->nin_flt + sig->cnin_flt;
psig->cnaj_flt +=
    p->naj_flt + sig->naj_flt + sig->cnaj_flt;
psig->cninblock +=
    task_io_get_inblock(p) +
    sig->inblock + sig->cninblock;
psig->cnoublock +=
    task_io_get_oublock(p) +
    sig->oublock + sig->cnoublock;
maxrss = max(sig->maxrss, sig->cnmaxrss);
if (psig->cnmaxrss < maxrss)
    psig->cnmaxrss = maxrss;
psig->cnvcs =
    p->nvcsw + sig->nvcsw + sig->cnvcs;
psig->cnivcs =
    p->nivcs + sig->nivcs + sig->cnivcs;
//To record the number of voluntary and involuntary context switches for both the parent process and the current process by adding the
//components together
task_io_accounting_add(&psig->ioac, &p->ioac);
task_io_accounting_add(&psig->ioac, &sig->ioac);
write_sequnlock(&psig->stats_lock);
spin_unlock_irq(&current->sigband->siglock);
}

if (wo->wo_rusage)
    getrusage(p, RUSAGE_BOTH, wo->wo_rusage);
status = (p->signal->flags & SIGNAL_GROUP_EXIT)

```

C Tab Width: 8 Ln 1125, Col 38 INS

Figure 6: Updating the wait_task_zombie() function in exit.c to account for the context switches in both the parent and the child process

```

Open  [icon]  *sys.c  Save  [icon] [icon] [icon]
~/Documents/Kernel copy/Kernel/Kernel

*sys.c  *exit.c  *core.c  *sched.h  *resource.h  *signal.h

}

goto out;

if (!lock_task_sigband(p, &flags))
    return;

switch (who) {
case RUSAGE_BOTH:
case RUSAGE_CHILDREN:
    utime = p->signal->cutime;
    stime = p->signal->cstime;
    r->ru_ninflt = p->signal->cnin_flt;
    r->ru_najflt = p->signal->cnaj_flt;
    r->ru_inblock = p->signal->cninblock;
    r->ru_oublock = p->signal->cnoublock;
    maxrss = p->signal->cnmaxrss;
    //To record the number of context switches from child processes
    r->ru_nvcsw = p->signal->cnvcs; //Voluntary context switches
    r->ru_nivcs = p->signal->cnivcs; //Involuntary context switches
    if (who == RUSAGE_CHILDREN)
        break;
case RUSAGE_SELF:
    thread_group_cputime_adjusted(p, &tgttime, &tgtstime);
    utime += tgttime;
    stime += tgttime;
    r->ru_ninflt += p->signal->nin_flt;
    r->ru_najflt += p->signal->naj_flt;
    r->ru_inblock += p->signal->inblock;
    r->ru_oublock += p->signal->oublock;
    if (maxrss < p->signal->maxrss)
        maxrss = p->signal->maxrss;
    //To record the number of context switches from parent processes
    r->ru_nvcsw += p->signal->nvcsw; //Voluntary context switches
    r->ru_nivcs += p->signal->nivcs; //Involuntary context switches
    t = p;
    do {
        accumulate_thread_rusage(t, r);
    } while_each_thread(p, t);
    break;
default:
    BUG();
}
unlock_task_sigband(p, &flags);

out:
r->ru_utime = ns_to_ttimeval(utime);
r->ru_stime = ns_to_ttimeval(stime);

```

C Tab Width: 8 Ln 1713, Col 34 INS

Figure 7: The changes to the getrusage() system call to record the context switches in the parent and child processes

The `getrusage()` system call is the next to be modified as shown above. It is modified to hold the context switches from either the parent or child process accordingly. The values are stored in the corresponding struct `rusage` that is passed and then can be accessed by the user from thereon as per requirements.

The kernel is then built and installed accordingly. Once completed, the system call is then tested with the help of the implementation program attached as a part of the package to test the number of context switches that occur in every thread instance of the said program.

Implementation Program

The mechanism for controlling access to the bathroom as specified in the description is implemented as a module in C — that is, as an interface file and an accompanying program module in the form of one or more `.c` files. Since we are dealing with only a single bathroom, the program file implements a global (i.e., compile-time) bathroom object, including synchronization primitives, to maintain its state and support waiting by an arbitrary number of users. It must keep track of how many people are currently in the bathroom and of which gender.

The main program is called `Sim_queue`. It takes a command line containing four arguments in the following format:— `Sim_queue nUsers meanLoopCount meanArrival meanStay`

In this command line

- **nUsers** is the number of people in the simulation. They shall be randomly distributed between men and women with equal probability.

- **meanLoopCount** is the average number of times that a user repeats the bathroom usage loop. The actual number of loop iterations is never be less than one.

- **meanArrival** is the mean of the arrival intervals. That is, each user will arrive at the bathroom an average of `meanArrival` units of time after the start of the simulation and also after leaving the bathroom from the previous use. Each thread generates an actual arrival time for each iteration from a normal distribution based on this mean.

- **meanStay** is the average length of time that a person stays in the bathroom, once he or she has entered. Stay times is generated from a normal distribution with this mean.

The compilation for this implementation program is done as follows:

```
gcc -g -c bathroom.c -Wall -pthread
gcc -g -c Sim_queue.c -Wall -pthread
gcc Sim_queue.o bathroom.o -o Sim_queue -Wall -pthread -lm
```

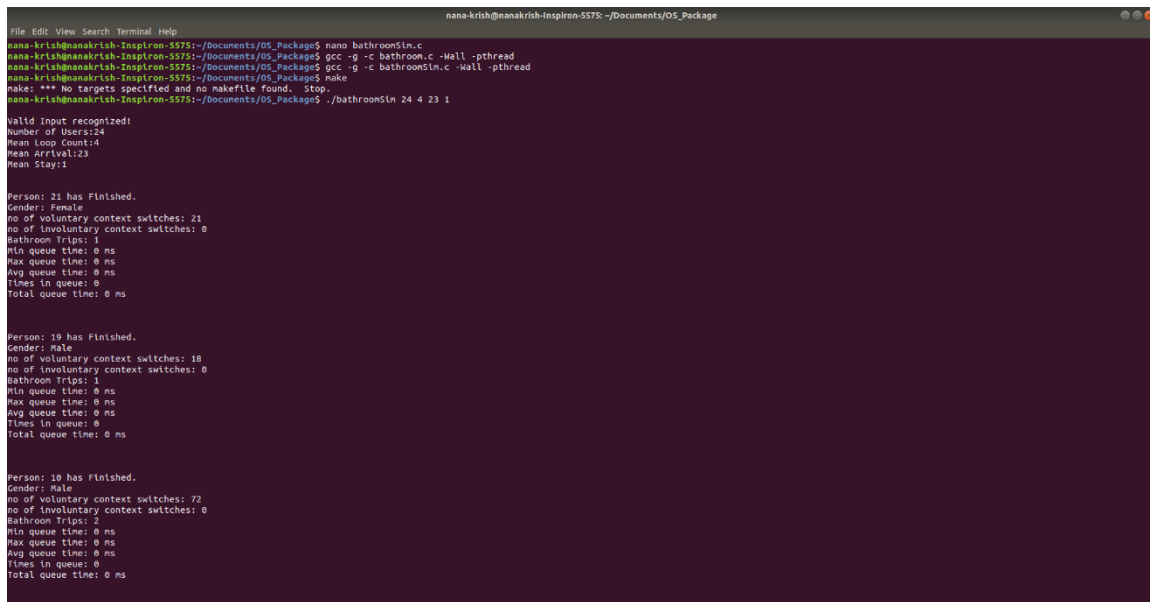
To run this program, the following steps are to be followed:

```
make
```

```
./Sim_queue nUsers meanLoopCount meanArrival meanStay
```

Where none of the four arguments are negative, over 1000 or not a number.

Results and Discussions



```
nana-krish@manakrish-Inspiron-5575: ~/Documents/OS_Package
File Edit View Search Terminal Help
nana-krish@manakrish-Inspiron-5575:~/Documents/OS_Package$ nano bathroomSim.c
nana-krish@manakrish-Inspiron-5575:~/Documents/OS_Package$ gcc -g -c bathroomSim.c -Wall -pthread
nana-krish@manakrish-Inspiron-5575:~/Documents/OS_Package$ gcc -g -c bathroomSim.c -Wall -pthread
nana-krish@manakrish-Inspiron-5575:~/Documents/OS_Package$ make
make: *** No targets specified and no makefile found.  Stop.
nana-krish@manakrish-Inspiron-5575:~/Documents/OS_Package$ ./bathroomSim 24 4 23 1

Valid Input recognized!
Number of Users:24
Mean Loop Count:4
Mean Arrival:23
Mean Stay:1

Person: 21 has Finished.
Gender: Female
no of voluntary context switches: 21
no of involuntary context switches: 0
Bathroom Trips: 1
Min queue time: 0 ms
Max queue time: 0 ms
Avg queue time: 0 ms
Times in queue: 0
Total queue time: 0 ms

Person: 19 has Finished.
Gender: Male
no of voluntary context switches: 18
no of involuntary context switches: 0
Bathroom Trips: 1
Min queue time: 0 ms
Max queue time: 0 ms
Avg queue time: 0 ms
Times in queue: 0
Total queue time: 0 ms

Person: 18 has Finished.
Gender: Male
no of voluntary context switches: 72
no of involuntary context switches: 0
Bathroom Trips: 2
Min queue time: 0 ms
Max queue time: 0 ms
Avg queue time: 0 ms
Times in queue: 0
Total queue time: 0 ms
```

Figure 8: The implementation program is run with custom arguments that satisfy set conditions

As is shown in Fig.8, the implementation program is compiled and run with a custom set of arguments passed on the command line as per requirement. Once done, the inputs are first verified and then the details of each thread, in this case, each person using the bathroom is displayed, along with which is included, the ID of the person, their Gender, the number of voluntary and involuntary context switches which is the prime requirement of this outcome, the number of times the person visited the bathroom, the minimum queue time, the maximum queue time, the average queue time and the total queue time.

```
File Edit View Search Terminal Help
nana.krish@nana.krish-inspiron-5575: ~/Documents/OS_Package

Times in queue: 1
Total queue time: 1 ms

Person: 13 has Finished.
Gender: Female
no of voluntary context switches: 758
no of involuntary context switches: 0
Bathroom trips: 5
Min queue time: 0 ms
Max queue time: 0 ms
Avg queue time: 0 ms
Times in queue: 0
Total queue time: 0 ms

Person: 16 has Finished.
Gender: Male
no of voluntary context switches: 1274
no of involuntary context switches: 0
Bathroom trips: 9
Min queue time: 0 ms
Max queue time: 0 ms
Avg queue time: 0 ms
Times in queue: 0
Total queue time: 0 ms

Person: 22 has Finished.
Gender: Male
no of voluntary context switches: 1136
no of involuntary context switches: 0
Bathroom trips: 7
Min queue time: 0 ms
Max queue time: 0 ms
Avg queue time: 0 ms
Times in queue: 0
Total queue time: 0 ms

Bathroom Statistics:
Total time of Simulation: 1752 ms
Total users: 200
Total times people were in the Queue:: 20 ms
Total time spent in Queue:: 0 ms
Minimum Time spent in Queue: 0 ms
Maximum Time spent in Queue: 1 ms
Avg time spent in Queue: 0.380000 ms
Avg Users in the Bathroom at any time: 0.050582
```

Figure 9: Overall end statistics being displayed at the end of the run

Furthermore, at the end of the program run, the overall usage statistics are also provided including total time of simulation, number of times the bathroom was used, total times people were in a queue, average time spent in a queue and so on and so forth.

Conclusion

The primary objective of the project, which was to extend the functionality of the `getrusage()` system call to record the number of context switches is achieved by making

1. Appropriate structural changes to the definition of the structures that record the values of individual processes and
2. Appropriate functional changes to the system call definition itself and its related dependencies in the kernel module and built accordingly.

Furthermore, the implementation program was chosen with the intention of managing a large number of threads in a single code structure so that a sizeable number of context switches may occur in the process of synchronizing the threads and thus, the values were recorded and displayed to suit the required outcome as expected.

Bibliography

1. <https://web.cs.wpi.edu/~cs3013/c07/Projects/Project3.pdf>
2. [https://docs.microsoft.com/en-us/previous-versions/windows/it-pro/windows-2000-server/cc938606\(v=technet.10\)](https://docs.microsoft.com/en-us/previous-versions/windows/it-pro/windows-2000-server/cc938606(v=technet.10))
3. <https://wiki.ubuntu.com/Kernel/BuildYourOwnKernel>
4. <https://web.cs.wpi.edu/~cs3013/a09/ProjectAssignments--a09/Project%203,%20Threads.htm>
5. [https://simple.wikipedia.org/wiki/C_\(programming_language\)](https://simple.wikipedia.org/wiki/C_(programming_language))
6. <https://pubs.opengroup.org/onlinepubs/009695399/functions/getrusage.html>
7. <https://www.geeksforgeeks.org/understanding-exit-abort-and-assert/>