CS3103 Tutorial

Tutorial 1: Getting Started with Linux

`ls`

`ls` is a command to list files and directories in the current directory.

Arguments:

`-l`: long listing format

```
ls -l
-rwxr-xr-x 1 user user 16384 Jan 1 2023 a.out
```

Remarks:

- `-`: file type (`-` = file, `d` = directory, `l` = symbolic link)
- `rwxr-xr-x`: file permissions
- `1`: number of hard links (number of directory entries that refer to the file)
- `user user`: owner and group
- `16384`: file size in bytes
- `Jan 1 2023`: last modified date
- `a.out`: file name

`-a`: list all files including hidden files starting with `.`

(This includes current directory `.` and parent directory `..`)

The following arguments should be combined with `-l`:

`-h`: human-readable file sizes

```
ls -lh
-rwxr-xr-x 1 user user 16K Jan 1 2023 a.out
```

`-S`: sort by file size, largest first

`-t`: sort by modification time, newest first

`-r`: reverse order while sorting

`-R`: list subdirectories recursively

Other commands

Command	Description	Remarks
`pwd`	print working directory	`~` = home directory = `/home/{username}`
`cd`	change directory	`cd` = `cd ~`

Command	Description	Remarks
`mkdir`	make directory	`-p`: create parent directories if they do not exist
`rmdir`	remove directory	directory must be empty
`cp`	copy files and directories	`-r`: copy directories recursively
`mv`	move files and directories	if move under the same directory, it is equivalent to renaming
`man`	display the manual page of a command	`--help` can be used as an alternative
`rm`	remove files	`-r`: remove directories recursively `-d`: remove directories `-f`: force remove without prompt

Remarks:

- `cp path1/file1 path2/file2`: copy `file1` to `path2` and rename it to `file2`
- `cp path1/file1 path2`: copy `file1` to `path2` and keep the same name
- `rm` can only remove files, not directories
- `rm -d` can only remove empty directories
- `rm -rd` can remove directories recursively, regardless of whether they are empty

Tutorial 3: Threads

`.s` file stands for assembly code.

Instruction	Description
`mov imm, $reg`	store immediate value into register
`mov mem, $reg`	load from memory into register
`mov $reg1, $reg2`	move from reg1 to reg2
`mov $reg, mem`	store from register into memory
`mov imm, mem`	store immediate value into memory
`add imm, $reg`	`reg += imm`
`add $reg1, $reg2`	`reg2 += reg1`
`sub imm, $reg`	`reg -= imm`
`sub $reg1, $reg2`	`reg2 -= reg1`

`test imm, $reg`
`test $reg, imm`
`test $reg1, $reg2`
	Compare two operands as specified immediately after `test`

Instruction	Description

`jne`
`je`
`jg`
`jge`
`jl`
`jle`
	Jump if not equal
	Jump if equal
	Jump if greater
	Jump if greater or equal
	Jump if less
	Jump if less or equal

`nop`	No operation

Example 1

```
.main
.top
sub  $1,%dx
test $0,%dx
jgte .top
halt
```

This assembly code does two things:

1. Decrement `%dx` by 1
2. If `%dx` is greater than or equal to 0, jump to `.top`; otherwise, halt

Therefore it will halt when `%dx` becomes negative.

We use a `x86.py` simulator to simulate the execution of x86 assembly code.

```
./x86.py -p loop.s -t 1 -i 100 -R dx
```

This command runs `loop.s` with 1 thread, an interrupt interval of 100, and trace the value of `%dx`.

The value of `%dx` will be 0 at start, and then it changes to -1, after which the program halts.

```
./x86.py -p loop.s -t 2 -i 100 -a dx=3,dx=3 -R dx
```

This command runs `loop.s` with 2 threads, an interrupt interval of 100, and trace the value of `%dx`. For each thread, `%dx` is initialized to 3.

For each thread, the value of `%dx` will be 3 at start, and then it changes to 2, 1, 0, -1, after which the program halts.

Since an interrupt happens every 100 instructions, no race condition occurs.

Example 2

```
.main
.top
# critical section
mov 2000, %ax  # get 'value' at address 2000
add $1, %ax    # increment it
mov %ax, 2000  # store it back

# see if we're still looping
sub  $1, %bx
test $0, %bx
jgt .top

halt
```

This assembly code consists of a critical section and a loop.

Critical section: Add 1 to the value at address 2000.

Loop: Decrement `%bx` by 1. If `%bx` is greater than 0, jump to `.top`; otherwise, halt.

So it loops `%bx` times, and in each loop, it increments the value at address 2000 by 1.

```
./x86.py -p looping-race-nolock.s -t 1 -M 2000
```

This command runs `loop-race-nolock.s` with 1 thread and trace the value at address 2000.

Since `%bx` is not initialized (0 by default), the program will halt after the first loop.

The value at address 2000 will be 0 at start, and then it changes to 1.

```
./x86.py -p looping-race-nolock.s -t 2 -M 2000 -i 4 -r -s 0
```

This command introduces 2 threads, an interrupt interval of 4, and a random seed of 0.

Since both threads are accessing the same memory location, a race condition occurs and the value is non-deterministic.

While it is expected to be 2, an interrupt occurring during the critical section will cause the result to be incorrect.

The critical section is:

```
mov 2000, %ax
add $1, %ax
mov %ax, 2000
```

If an interrupt occurs during the critical section, the result will be incorrect. Otherwise, the result will be 2.

```
./x86.py -p looping-race-nolock.s -a bx=1 -t 2 -M 2000 -i 1
```

This commands introduces 2 threads, initializes `%bx` to 1, and an interrupt interval of 1.

As analyzed above, an interrupt interval shorter than the instructions in the critical section will cause the result to be incorrect.

`-i 1` results in a value of 1, which is incorrect.

`-i 2` results in a value of 1, which is incorrect.

`-i 3` results in a value of 2, which is correct.

An interrupt interval of 3 or above should give the correct result.

```
./x86.py -p looping-race-nolock.s -a bx=100 -t 2 -M 2000 -i ?
```

This commands introduces 2 threads, initializes `%bx` to 100, and a fixed interrupt interval.

Since the critical section has a length of 3 instructions, an interrupt interval which is a multiple of 3 should give the correct result.

On the other hand, if the interrupt interval avoids interruptions during the critical section for both threads, the result will also be correct. In this case, to guarantee correctness, the interrupt interval should be at least 597 (which is the length of 99 loops + 1 critical section).

Therefore, the interrupt interval should be either a multiple of 3 or at least 597.

For other interrupt intervals, the result will be incorrect. These include all positive integers up to 596 that are not multiples of 3.

The result will be a value less than 200 (e.g. -i 1 yields 100, -i 2 yields 100, -i 4 yields 150).

Tutorial 4: CPU Scheduling

We use a `scheduler.py` simulator to simulate CPU scheduling.

```
./scheduler.py -p FIFO -j 3 -s 100
```

Parameters:

- `-p` : policy (FIFO, RR, SJF (Preemptive))
- `-j` : number of jobs
- `-s` : seed (will be used to generate random job lengths)
- `-l` : length of each job

The above example runs the FIFO policy with 3 jobs and a random seed of 100.

The job lengths are randomly generated to be 2, 5 and 8, respectively.

The result is:

```
Final statistics:
  Job   0 -- Response: 0.00  Turnaround 2.00  Wait 0.00
  Job   1 -- Response: 2.00  Turnaround 7.00  Wait 2.00
  Job   2 -- Response: 7.00  Turnaround 15.00  Wait 7.00

  Average -- Response: 3.00  Turnaround 8.00  Wait 3.00
```

Example 1

```
./scheduler.py -p SJF -l 200,200,200
./scheduler.py -p FIFO -l 200,200,200
```

The above commands run the SJF and FIFO policies with 3 jobs, each with a length of 200.

Under SJF,

- Job 0: Response time = 0, Turnaround time = 200
- Job 1: Response time = 200, Turnaround time = 400
- Job 2: Response time = 400, Turnaround time = 600
- Average response time = 200, Average turnaround time = 400

Under FIFO,

- Job 0: Response time = 0, Turnaround time = 200
- Job 1: Response time = 200, Turnaround time = 400
- Job 2: Response time = 400, Turnaround time = 600
- Average response time = 200, Average turnaround time = 400

Example 2

```
./scheduler.py -p SJF -l 100,200,300
./scheduler.py -p FIFO -l 100,200,300
```

The above commands run the SJF and FIFO policies with 3 jobs, each with a length of 100, 200 and 300.

Under SJF,

- Job 0: Response time = 0, Turnaround time = 100
- Job 1: Response time = 100, Turnaround time = 300
- Job 2: Response time = 300, Turnaround time = 600
- Average response time = 133.33, Average turnaround time = 333.33

Under FIFO,

- Job 0: Response time = 0, Turnaround time = 100
- Job 1: Response time = 100, Turnaround time = 300
- Job 2: Response time = 300, Turnaround time = 600
- Average response time = 133.33, Average turnaround time = 333.33

If the order of the jobs is changed:

For SJF, the result will be the same. Since all jobs are ready at t=0, the shortest job will be executed first. Therefore, the execution order is always the same (100, 200, 300).

For FIFO, the result will be different. The jobs are executed in the order they arrive. Therefore, the execution order will be affected by the input order. Further, any other order will result in a larger average response time and turnaround time because 100, 200, 300 is the optimal order. The order 300, 200, 100 will result in the largest average response time of 266.67 and average turnaround time of 466.67.

Example 3

```
./scheduler.py -p RR -l 3,3,3 -q 1
./scheduler.py -p RR -l 3,2,4 -q 1
```

The above commands run the RR policy with 3 jobs with a time quantum of 1.

The 3, 3, 3 case:

- Job 0 will be executed at t=0~1, 3~4, 6~7, Response time = 0, Turnaround time = 7
- Job 1 will be executed at t=1~2, 4~5, 7~8, Response time = 1, Turnaround time = 8
- Job 2 will be executed at t=2~3, 5~6, 8~9, Response time = 2, Turnaround time = 9
- Average response time = 1, Average turnaround time = 8

The 3, 2, 4 case:

- Job 0 will be executed at t=0~1, 3~4, 6~7, Response time = 0, Turnaround time = 7
- Job 1 will be executed at t=1~2, 4~5, Response time = 1, Turnaround time = 5
- Job 2 will be executed at t=2~3, 5~6, 7~8, 8~9, Response time = 2, Turnaround time = 9
- Average response time = 1, Average turnaround time = 7

Analysis

1. For what types of workloads does SJF deliver the same turnaround times as FIFO?

Answer: In either of the following cases, SJF delivers the same turnaround times as FIFO:

- All jobs have the same burst time
- All jobs arrive at the exact time when a context switch occurs and they arrive in the same order as in FIFO, so that SJF never preemptively switches to another job

2. For what types of work loads and quantum lengths does SJF deliver the same response times as RR?

Answer: When all processes have the same burst time as the time quantum of RR, SJF delivers the same response times as RR.

3. What happens to response time with SJF as job lengths increase? Can you use the simulator to demonstrate the trend?

Answer: the response time increases as job lengths increase. Compare results of the following simulations:

```
./scheduler.py -p SJF -l 100,100,100
./scheduler.py -p SJF -l 200,200,200
./scheduler.py -p SJF -l 200,300,400
```

The 100, 100, 100 case: average response time = 100 (0 for job 0, 100 for job 1, 200 for job 2)

The 200, 200, 200 case: average response time = 200 (0 for job 0, 200 for job 1, 400 for job 2)

The 200, 300, 400 case: average response time = 233.33 (0 for job 0, 200 for job 1, 500 for job 2)

4. What happens to response time with RR as quantum lengths increase? Can you write an equation that gives the worst-case response time, given N jobs?

Answer: the response time increases as quantum lengths increase.

The equation is `worst_case_response_time = time_quantum * sum(i for i in range(N))` which can be simplified to `worst_case_response_time = time_quantum * N * (N - 1) / 2`.

Tutorial 5: Condition Variables

In this tutorial, 3 examples programs are given to resolve the producer-consumer problem using condition variables.

(1) `main-one-cv-while.c`: A program that uses one condition variable and a while statement to synchronize threads.

(2) `main-two-cvs-if.c`: A program that uses two condition variables and an if statement to synchronize threads.

(3) `main-two-cvs-while.c`: A program that uses two condition variables and a while statement to synchronize threads. (This is the correct way to use condition variables.)

Code

(1) `main-one-cv-while.c`

```
pthread_cond_t cv = PTHREAD_COND_INITIALIZER;
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;

void *producer(void *arg) {
    int id = (int) arg;
    // make sure each producer produces unique values
    int base = id * loops;
    int i;
    for (i = 0; i < loops; i++) {   p0;
       Mutex_lock(&m);              p1;
       while (num_full == max) {    p2;
           Cond_wait(&cv, &m);      p3;
       }
       do_fill(base + i);           p4;
       Cond_signal(&cv);            p5;
       Mutex_unlock(&m);            p6;
    }
    return NULL;
}

void *consumer(void *arg) {
    int id = (int) arg;
    int tmp = 0;
    int consumed_count = 0;
    while (tmp != END_OF_STREAM) { c0;
       Mutex_lock(&m);             c1;
       while (num_full == 0) {     c2;
           Cond_wait(&cv, &m);     c3;
        }
       tmp = do_get();             c4;
       Cond_signal(&cv);           c5;
       Mutex_unlock(&m);           c6;
       consumed_count++;
    }
```

```
    // return consumer_count-1 because END_OF_STREAM does not count
    return (void *) (long long) (consumed_count - 1);
}
```

(2) `main-two-cvs-if.c`

```
void *producer(void *arg) {
    int id = (int) arg;
    // make sure each producer produces unique values
    int base = id * loops;
    int i;
    for (i = 0; i < loops; i++) {   p0;
       Mutex_lock(&m);              p1;
       if (num_full == max) {       p2;
           Cond_wait(&empty, &m);   p3;
       }
       do_fill(base + i);           p4;
       Cond_signal(&fill);          p5;
       Mutex_unlock(&m);            p6;
    }
    return NULL;
}

void *consumer(void *arg) {
    int id = (int) arg;
    int tmp = 0;
    int consumed_count = 0;
    while (tmp != END_OF_STREAM) { c0;
       Mutex_lock(&m);              c1;
       if (num_full == 0) {         c2;
           Cond_wait(&fill, &m);    c3;
        }
       tmp = do_get();              c4;
       Cond_signal(&empty);         c5;
       Mutex_unlock(&m);            c6;
       consumed_count++;
    }

    // return consumer_count-1 because END_OF_STREAM does not count
    return (void *) (long long) (consumed_count - 1);
}
```

(3) `main-two-cvs-while.c`

```
void *producer(void *arg) {
    int id = (int) arg;
    // make sure each producer produces unique values
    int base = id * loops;
    int i;
    for (i = 0; i < loops; i++) {   p0;
       Mutex_lock(&m);              p1;
       while (num_full == max) {    p2;
           Cond_wait(&empty, &m);   p3;
```

```
        }
        do_fill(base + i);          p4;
        Cond_signal(&fill);         p5;
        Mutex_unlock(&m);           p6;
    }
    return NULL;
}

void *consumer(void *arg) {
    int id = (int) arg;
    int tmp = 0;
    int consumed_count = 0;
    while (tmp != END_OF_STREAM) { c0;
        Mutex_lock(&m);             c1;
        while (num_full == 0) {     c2;
            Cond_wait(&fill, &m);   c3;
         }
        tmp = do_get();             c4;
        Cond_signal(&empty);        c5;
        Mutex_unlock(&m);           c6;
        consumed_count++;
    }

    // return consumer_count-1 because END_OF_STREAM does not count
    return (void *) (long long) (consumed_count - 1);
}
```

Example 1

```
./main-two-cvs-while -l 3 -m 2 -p 1 -c 1 -v
```

Parameters:

- `-l 3`: each producer produces 3 items
- `-m 2`: the buffer size is 2
- `-p 1`: 1 producer
- `-c 1`: 1 consumer
- `-v`: verbose mode

Tutorial 6: Deadlock

1. First let's make sure you understand how the programs generally work, and some of the key options. Study the code in vector-deadlock.c, as well as in main-common.c and related files. Now, run ./vector-deadlock -d -n 2 -l 1 -t, which instantiates two threads (-n 2), each of which does one vector add (-l 1). Change the number of loops (-l) from 1 to higher numbers. What happens? Does the code (always) deadlock?

(1) `vector-deadlock.c`

```
void vector_add(vector_t *v_dst, vector_t *v_src) {
    Pthread_mutex_lock(&v_dst->lock);
    Pthread_mutex_lock(&v_src->lock);
    int i;
    for (i = 0; i < VECTOR_SIZE; i++) {
        v_dst->values[i] = v_dst->values[i] + v_src->values[i];
    }
    Pthread_mutex_unlock(&v_dst->lock);
    Pthread_mutex_unlock(&v_src->lock);
}

void fini() {}
```

(2) `main-common.c` (part of main function)

```
    pthread_t pid[MAX_THREADS];
    thread_arg_t args[MAX_THREADS];
    int i;
    for (i = 0; i < num_threads; i++) {
        args[i].tid = i;
        if (enable_parallelism == 0) {
            args[i].vector_0 = 0;
            args[i].vector_1 = 1;
        } else {
            args[i].vector_0 = i * 2;
            args[i].vector_1 = i * 2 + 1;
        }

        if (cause_deadlock && i % 2 == 1)
            args[i].vector_add_order = 1;
        else
            args[i].vector_add_order = 0;
    }

    for (i = 0; i < 2 * MAX_THREADS; i++)
        vector_init(&v[i], i);

    double t1 = Time_GetSeconds();

    for (i = 0; i < num_threads; i++)
        Pthread_create(&pid[i], NULL, worker, (void *) &args[i]);
    for (i = 0; i < num_threads; i++)
        Pthread_join(pid[i], NULL);

    double t2 = Time_GetSeconds();

    fini();
    if (do_timing) {
        printf("Time: %.2f seconds\n", t2 - t1);
    }
```

Answer: When the number of loop is 1, there is no deadlock. When the number of loop is higher than 1, the code could possibly deadlock, but not always.

2. How does changing the number of threads (-n) change the outcome of the program? Are there any values of -n that ensure no deadlock occurs?

Answer: When n >= 2, the larger the number of threads, the more likely the code will deadlock. -n 1 ensures no deadlock occurs.

3. Now examine the code in vector-global-order.c. First, make sure you understand what the code is trying to do; do you understand why the code avoids deadlock? In the vector_add() routine, there are two cases to consider: when the source and destination vectors are different and when they are the same. What is the difference between these two scenarios, i.e., why we should consider the later one?

(3) `vector-global-order.c`

```
void vector_add(vector_t *v_dst, vector_t *v_src) {
    if (v_dst < v_src) {
        Pthread_mutex_lock(&v_dst->lock);
        Pthread_mutex_lock(&v_src->lock);
    } else if (v_dst > v_src) {
        Pthread_mutex_lock(&v_src->lock);
        Pthread_mutex_lock(&v_dst->lock);
    } else {
        // special case: src and dst are the same
        Pthread_mutex_lock(&v_src->lock);
    }
    int i;
    for (i = 0; i < VECTOR_SIZE; i++) {
        v_dst->values[i] = v_dst->values[i] + v_src->values[i];
    }
    Pthread_mutex_unlock(&v_src->lock);
    if (v_dst != v_src)
        Pthread_mutex_unlock(&v_dst->lock);
}

void fini() {}
```

Answer: The code avoids deadlock because it locks the mutexes in a global order. In other words, the mutex with the smaller address is locked first, which is guaranteed to be the same order across all threads.

The special case is considered because if the source and destination vectors are the same, the code will try to acquire the same lock twice, resulting in a deadlock.

4. Now run the code with the following flags: -t -n 2 -l 100000 -d. How long does the code take to complete? How does the total time change when you increase the number of loops, or the number of threads?

Answer: The code takes around 0.06 seconds.

With the number of threads fixed to 2, the number of loops taking values 100000, 1000000 and 10000000 result in times of 0.06, 0.42, 3.54 seconds respectively. Therefore, the total time increases approximately linearly with the number of loops.

With the number of loops fixed to 100000, the number of threads taking values 4, 8, 16, 32 and 64 result in times of 0.25, 0.55, 1.13, 2.19, 4.48 seconds respectively. Therefore, the total time increases approximately linearly with the number of threads.

5. What happens if you turn on the parallelism flag (-p)? How much will the performance (i.e., total time) change when each thread is working on adding different vectors (which is what -p enables) versus working on the same ones?

Answer: When each thread is working on adding different vectors, the threads no longer need to wait for locks, so the performance improves.

The total time decreases significantly. For example, -n 64 -l 100000 -d -p results in a time of 0.75 seconds. -n 2 -l 10000000 -d -p results in a time of 1.70 seconds, which is much faster than the time without parallelism.

It can also be noted that the parallelism improves the performance more significantly when the number of threads is larger, compared to when the number of loops is larger.

6. Now let's study vector-try-wait.c. First make sure you understand the code. Is the first call to pthread_mutex_trylock() really needed? Now run the code. How fast does it run compared to the global order approach? How does the number of retries, as counted by the code, change as the number of threads increases?

(4) `vector-try-wait.c`

```
void vector_add(vector_t *v_dst, vector_t *v_src) {
  top:
    if (pthread_mutex_trylock(&v_dst->lock) != 0) {
        goto top;
    }
    if (pthread_mutex_trylock(&v_src->lock) != 0) {
        retry++;
        Pthread_mutex_unlock(&v_dst->lock);
        goto top;
    }
    int i;
    for (i = 0; i < VECTOR_SIZE; i++) {
        v_dst->values[i] = v_dst->values[i] + v_src->values[i];
    }
    Pthread_mutex_unlock(&v_dst->lock);
    Pthread_mutex_unlock(&v_src->lock);
}
```

The first call to pthread_mutex_trylock() is not needed because use pthread_mutex_lock(&v_dst->lock) will put the thread to sleep if the lock is not available, saving CPU cycles. However, if a trylock is used, the thread will keep trying to acquire the lock, making it less efficient.

./vector-try-wait -n 2 -l 100000 -d takes 0.25 seconds. The performance is worse than the global order approach (0.06 seconds).

With the number of loops fixed to 100000, the number of threads taking values 2, 4 and 6 results in 1056982, 3032613, 7012497 retries respectively. Therefore, the number of retries increases approximately exponentially with the number of threads.

7. Now let's look at vector-avoid-hold-and-wait.c. What is the main problem with this approach? How does its performance compare to the other versions (i.e., vector-global-order and vector-try-wait), when running both with -p and without it?

(5) `vector-avoid-hold-and-wait.c`

```
void vector_add(vector_t *v_dst, vector_t *v_src) {
    // put GLOBAL lock around all lock acquisition...
    Pthread_mutex_lock(&global);
    Pthread_mutex_lock(&v_dst->lock);
    Pthread_mutex_lock(&v_src->lock);
    Pthread_mutex_unlock(&global);
    int i;
    for (i = 0; i < VECTOR_SIZE; i++) {
        v_dst->values[i] = v_dst->values[i] + v_src->values[i];
    }
    Pthread_mutex_unlock(&v_dst->lock);
    Pthread_mutex_unlock(&v_src->lock);
}

void fini() {}
```

The problem with this approach is that it uses a global lock to prevent deadlock. This is unnecessary and inefficient because it forces all threads to wait for the global lock, even if they are accessing different vectors.

./vector-avoid-hold-and-wait -t -n 2 -l 100000 -d takes 0.11 seconds. With -p, it takes 0.05 seconds.

./vector-avoid-hold-and-wait -t -n 8 -l 100000 -d takes 0.86 seconds. With -p, it takes 0.52 seconds.

./vector-avoid-hold-and-wait -t -n 16 -l 100000 -d takes 1.97 seconds. With -p, it takes 1.87 seconds.

It can be concluded that the performance of running vector-avoid-hold-and-wait with -p is just slightly better than running it without -p.

Compared to other versions, vector-avoid-hold-and-wait performs worse than vector-global-order and vector-try-wait under parallelism. When running without -p, vector-avoid-hold-and-wait performs better than vector-try-wait but worse than vector-global-order.

8. Finally, let's look at vector-nolock.c. This version doesn't use locks at all; does it provide the exact same semantics as the other versions? Why or why not?

(6) `vector-nolock.c`

```
int fetch_and_add(int * variable, int value) {
    asm volatile("lock; xaddl %%eax, %2;"
                 :"=a" (value)
                 :"a" (value), "m" (*variable)
                 :"memory");
    return value;
}

void vector_add(vector_t *v_dst, vector_t *v_src) {
    int i;
    for (i = 0; i < VECTOR_SIZE; i++) {
        fetch_and_add(&v_dst->values[i], v_src->values[i]);
    }
}

void fini() {}
```

It uses fetch-and-add to atomically add values to the destination vector.

However, this version does not provide the exact same semantics as the other versions. Other versions treat the process of adding every pair of values as a critical section, which makes it an atomic operation. In contrast, vector-nolock treats adding a pair of values as an atomic operation, so that it cannot guarantee that the entire process of adding all pairs of values is not interrupted by other threads.

9. Now compare its performance to the version vector-global-order, both when threads are working on the same two vectors (no -p) and when each thread is working on separate vectors (-p). How does this no-lock version perform?

./vector-nolock -t -n 2 -l 100000 -d takes 0.45 seconds. With -p, it takes 0.09 seconds.

./vector-nolock -t -n 4 -l 100000 -d takes 1.07 seconds. With -p, it takes 0.09 seconds.

./vector-nolock -t -n 8 -l 100000 -d takes 6.02 seconds. With -p, it takes 0.15 seconds.

With high contention (no -p), the performance of vector-nolock is worse than vector-global-order. However, with high parallelism (-p), the performance of vector-nolock is much better than vector-global-order.

Tutorial 7: Paging

1. How should page-table size change as the address space grows? As the page size grows? Why not use big pages in general?

```
-P 1k -a 1m -p 512m -v -n 0
-P 1k -a 2m -p 512m -v -n 0
-P 1k -a 4m -p 512m -v -n 0

-P 1k -a 1m -p 512m -v -n 0
-P 2k -a 1m -p 512m -v -n 0
-P 4k -a 1m -p 512m -v -n 0
```

(1) The page-table size is proportional to the address space. This is because number of pages = address space / page size. In the example, page size is fixed at 1KB, address space varies from 1MB, 2MB to 4MB, and the number of pages is 1024, 2048, 4096 respectively.

(2) The page-table size is inversely proportional to the page size. In the example, address space is fixed at 1MB, page size varies from 1KB, 2KB to 4KB, and the number of pages is 1024, 512, 256 respectively.

(3) Big pages are not used in general because internal fragmentation increases with big pages. This is because the last page allocated to a process may not be fully utilized, leading to wasted memory.

2. What happens as you increase the percentage of pages that are allocated in each address space?

```
-P 1k -a 16k -p 32k -v -u 0
-P 1k -a 16k -p 32k -v -u 25
-P 1k -a 16k -p 32k -v -u 50
-P 1k -a 16k -p 32k -v -u 75
-P 1k -a 16k -p 32k -v -u 100
```

More pages are valid. This is because when more pages are allocated in each address space, the percentage of page faults caused by invalid references decreases.

3. Which of these parameter combinations are unrealistic? Why?

```
-P 8 -a 32 -p 1024 -v -s 1
-P 8k -a 32k -p 1m -v -s 2
-P 1m -a 256m -p 512m -v -s 3
```

-P 8 -a 32 -p 1024 -v -s 1 is unrealistic because address space = 32 bytes, page size = 8 bytes, physical memory = 1024 bytes. There are 1024/8 = 128 entries in each page table, but the address space can only hold 32/8 = 4 entries. These parameters are also too small to be realistic.

-P 8k -a 32k -p 1m -v -s 2 is unrealistic because address space = 32 KB, page size = 8 KB, physical memory = 1 MB. There are 1 MB / 8 KB = 128 entries in each page table, but the address space can only hold 32 KB / 8 KB = 4 entries. Therefore a process can be allocated at most 4 pages, which leads to significant internal fragmentation.

-P 1m -a 256m -p 512m -v -s 3 is unrealistic because page size = 1 MB is too large, compared to the typical value of 4 KB. This will lead to significant internal fragmentation.

Though, all of these parameter combinations can successfully run the simulator.