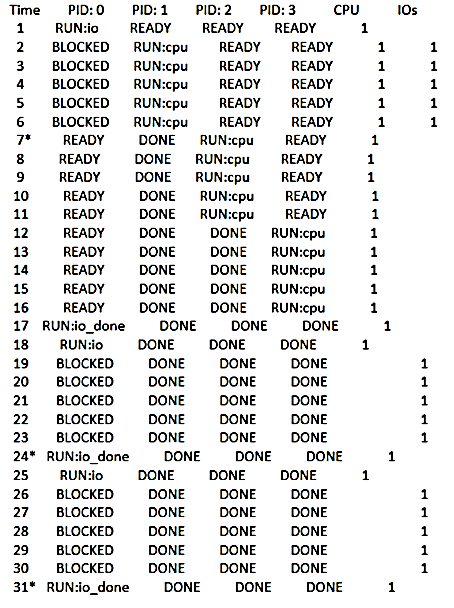
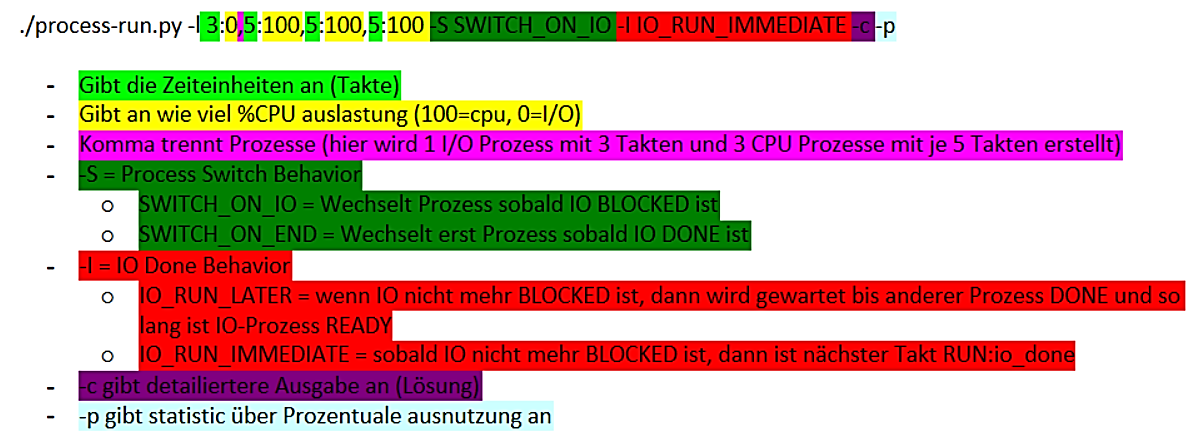
OSTEP

Homework



# Kapitel 4 – Process

****

1. Run process-run.py with the following flags: -l 5:100,5:100. What should the CPU utilization be (e.g., the percent of time the CPU is in use?) Why do you know this? Use the -c and -p flags to see if you were right.

* It takes 10-time units because every cpu instruction take 1 time unit. We pass 2 program arguments with 5 instructions each.

1. Now run with these flags: ./process-run.py -l 4:100,1:0. These flags specify one process with 4 instructions (all to use the CPU), and one that simply issues an I/O and waits for it to be done. How long does it take to complete both processes? Use -c and -p to find out if you were right.

* It takes 11-time units, because for the IO action we need 1 time unit for start, 5 for BLOCKED and 1 for done IO

1. Switch the order of the processes: -l 1:0,4:100. What happens now? Does switching the order matter? Why? (As always, use -c and -p to see if you were right)

* It takes 7-time units because the cpu can run with the second program while the first one is in BLOCKED state.

1. We’ll now explore some of the other flags. One important flag is -S, which determines how the system reacts when a process issues an I/O. With the flag set to SWITCH ON END, the system will NOT switch to another process while one is doing I/O, instead waiting until the process is completely finished. What happens when you run the following two processes (-l 1:0,4:100 -c -S SWITCH ON END), one doing I/O and the other doing CPU work?

* It will take 11-time units because we have to wait for the first instruction (the IO) to be finished.

1. Now, run the same processes, but with the switching behavior set to switch to another process whenever one is WAITING for I/O (-l 1:0,4:100 -c -S SWITCH ON IO). What happens now? Use -c and -p to confirm that you are right.

* Like in question 3, it switches to process 1 while process 0 is in BLOCKED state

1. One other important behavior is what to do when an I/O completes. With -I IO RUN LATER, when an I/O completes, the process that issued it is not necessarily run right away; rather, whatever was running at the time keeps running. What happens when you run this combination of processes? (Run ./process-run.py -l 3:0,5:100,5:100,5:100 -S SWITCH ON IO -I IO RUN LATER -c -p) Are system resources being effectively utilized?

* At first the process 1 can run while process 0 is in BLOCKED state. Because of the IO\_RUN\_LATER flag it doesn’t finishes process 0 after the blocked state is over, it starts with process 2 and after it process 3 and then it goes back to process 0 instruction 1 where it has to finish it before starting instruction 2 and 3 from process 0. It would be more effective to run the process 1, 2 and 3 in between the 3 IO instructions from process 0. We could save 10-time units.

1. Now run the same processes, but with -I IO RUN IMMEDIATE set, which immediately runs the process that issued the I/O. How does this behavior differ? Why might running a process that just completed an I/O again be a good idea?

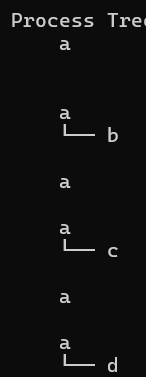
* IO instructions are now more prioritised we save 10-time units because every time an IO instruction is in progress and the state is blocked, the cpu can process the “normal” instructions from program 1, 2 and 3. The scheduling policy might consider to run another IO instruction because the current cpu workflow is effective (100% utilization)

1. Now run with some randomly generated processes: -s 1 -l 3:50,3:50 or -s 2 -l 3:50,3:50 or -s 3 -l 3:50,3:50. See if you can predict how the trace will turn out. What happens when you use the flag -I IO RUN IMMEDIATE vs. -I IO RUN LATER? What happens when you use -S SWITCH ON IO vs. -S SWITCH ON END?

* Flag SWITCH\_ON\_END always waits till process 0 is completely finished before switching to process 2
* Flag SWITCH\_ON\_IO (default) switches while IO is BLOCKED
* Flag IO\_RUN\_IMMEDIATE prioritises “IO” instruction
* Flag IO\_RUN\_LATER prioritises “normal cpu” instruction

# Kapitel 5 – Process API

**Questions:**

1. Run ./fork.py -s 10 and see which actions are taken. Can you predict what the process tree looks like at each step? Use the -c flag to check your answers. Try some different random seeds (-s) or add more actions (-a) to get the hang of it.

* If one branch forks another it becomes its child. If one branch exits it disappears.

1. One control the simulator gives you is the fork percentage, controlled by the -f flag. The higher it is, the more likely the next action is a fork; the lower it is, the more likely the action is an exit. Run the simulator with a large number of actions (e.g., -a 100) and vary the fork percentage from 0.1 to 0.9. What do you think the resulting final process trees will look like as the percentage changes? Check your answer with -c.

* The higher the fork-percentage is set, the bigger Tree of processes will be possible and appear. It is very likely that the lower the percentage is, the Tree will return to its root size of just one branch (a).

1. Now, switch the output by using the -t flag (e.g., run ./fork.py -t). Given a set of process trees, can you tell which actions were taken?

* this example oscillates between a forks b,c or d and a exits b, c or d.

1. One interesting thing to note is what happens when a child exits; what happens to its children in the process tree? To study this, let’s create a specific example: ./fork.py -A a+b,b+c,c+d,c+e,c-. This example has process ’a’ create ’b’, which in turn creates ’c’, which then creates ’d’ and ’e’. However, then, ’c’ exits. What do you think the process tree should like after the exit? What if you use the -R flag? Learn more about what happens to orphaned processes on your own to add more context.

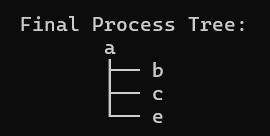
* I expected the sub-branches of c to be appended to b by default, but it was appended to the root a. With the -R flag, the expected result appears. The sub-branches append to the next lower branch b after leaving c.

1. One last flag to explore is the -F flag, which skips intermediate steps and only asks to fill in the final process tree. Run ./fork.py -F and see if you can write down the final tree by looking at the series of actions generated. Use different random seeds to try this a few times.

* -F skips the steps in between

1. Finally, use both -t and -F together. This shows the final process tree, but then asks you to fill in the actions that took place. By looking at the tree, can you determine the exact actions that took place? In which cases can you tell? In which can’t you tell? Try some different random seeds to delve into this question.

* For the seed 4 we have more than one solution to get to this Tree in 5 steps:



Opportunity 1: a forks b, a forks c, a forks d, d exits, a forks e

Opportunity 2: a forks b, a fors c, a forks d, a forks e, d exits (only with -R flag)

**Code:**

1. Write a program that calls fork(). Before calling fork(), have the main process access a variable (e.g., x) and set its value to something (e.g., 100). What value is the variable in the child process? What happens to the variable when both the child and parent change the value of x?

* I set x = 100. In the child process I sub 2 from x so we expect 98. In the parent process I add 2 we expect 102. X was defined and implemented before the fork call. So each process calculating with its own X (value = 100).

1. Write a program that opens a file (with the open() system call) and then calls fork() to create a new process. Can both the child and parent access the file descriptor returned by open()? What happens when they are writing to the file concurrently, i.e., at the same time?

* Both parent and child process accesses the same output (test.output) as expected.

1. Write another program using fork(). The child process should print “hello”; the parent process should print “goodbye”. You should try to ensure that the child process always prints first; can you do this without calling wait() in the parent?

* I did it with usleep() to guarantee that the child will execute first, possible solution could be to somehow communicate with the child process?

1. Write a program that calls fork() and then calls some form of exec() to run the program /bin/ls. See if you can try all of the variants of exec(), including (on Linux) execl(), execle(), execlp(), execv(), execvp(), and execvpe(). Why do you think there are so many variants of the same basic call?

* Execl and execle works not quite as expected. After printing child and parent output a prompt appears then its executing the “ls” call. After calling ls no prompt is shown, I have to quit with Command-C. The different the various exec versions offer different options for how (i.e as vector, as array, as String) the arguments, i.e. programme name and execution, are transmitted to exec.

1. Now write a program that uses wait() to wait for the child process to finish in the parent. What does wait() return? What happens if you use wait() in the child?

* It seems adding wait() to child process has no impact because it has no child to wait for. If wait() was successful it returns the PID number if it fails it returns -1.

1. Write a slight modification of the previous program, this time using waitpid() instead of wait(). When would waitpid() be useful?

* With waitpid() we can define on which child we should wait. This makes sense if we have a lot of different processes and we need to manage which specific process should run or not run.

1. Write a program that creates a child process, and then in the child closes standard output (STDOUT FILENO). What happens if the child calls printf() to print some output after closing the descriptor?

* After closing the standard output, the child process can no longer output anywhere, as no output is active. The message is lost.

1. Write a program that creates two children, and connects the standard output of one to the standard input of the other, using the pipe() system call.

* First creating a Pipe with to ends (input to write() and output to read()). Second pass something to transmit with write() into the pipe. Third read the pipe output from the second child. I imagine the pipe as a temporary save which can be accessed from different processes.

# Kapitel 7 – Scheduling

**Questions:**

scheduler.py

|  |  |
| --- | --- |
| -s SEED | the random seed |
| -j JOBS | randome jobs |
| -l JLIST | instead of random jobs, provide a comma-separated list of run times |
| -m MAXLEN | max length of job |
| -p POLICY | sched policy to use: SJF, FIFO, RR |
| -q QUANTUM | length of time slice for RR policy |
| - c | compute answers for me |

Bsp: prompt> ./scheduler.py -p SJF -l 5,10,15

1. Compute the response time and turnaround time when running three jobs of length 200 with the SJF and FIFO schedulers.

./scheduler.py -p [SJF/FIFO] -l 200,200,200

* T = ((200-0) + (400 -200) + (600 – 400)) / 3 = 200 // Wieso steht im Programm 400??
* R = ((0 – 0) + (200 – 0) + (400 – 0)) / 3 = 200
* Egal ob SJF oder FIFO weil die jobs gleich groß sind

1. Now do the same but with jobs of different lengths: 100, 200, and 300.

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

* T = ((100-0) + (300 -100) + (600 – 300)) / 3 = 200 // Wieso steht im Programm 333,33??
* R = ((0 – 0) + (100 – 0) + (300 – 0)) / 3 = 133
* Wieder Egal weil die jobs schon von kurz zu lang sortiert sind

1. Now do the same, but also with the RR scheduler and a time-slice of 1.

./scheduler.py -p RR -q 1 -l 100,200,300

* T = ((298-0) + (499 -1) + (600 – 2)) / 3 = 465
* R = ((1 – 0) + (2 – 0) + (3 – 0)) / 3 = 1

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

* Wenn die jobgröße schon der Größe nach sortiert ist

1. For what types of workloads and quantum lengths does SJF deliver the same response times as RR?

* Die Jobs müssten alle die gleiche größe haben und die quantum lengths so lang wie die Ausführungszeit eines jobs

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

* Sie wird linear größer, da es immer länger braucht bis die einzelnen jobs ihren firstrun haben

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

* Je höher die Quantum lengths ist, desto höher wird die response time weil der firstrun immer später kommt
* Worst-Case Response Time = (N - 1) \* Q

# Kapitel 8 – Multi-Level Feedback Queue (MLFQ)

**Questions:**

mlfq.py

|  |  |
| --- | --- |
| -s SEED | the random seed |
| -n NUMQUEUES | number of queues in MLFQ (if not using -Q) |
| -q QUANTUM | length of time slice (if not using -Q) |
| -a ALLOTMENT | length of allotment (if not using -A) |
| -Q QUANTUMLIST | length of time slice per queue level, specified as x,y,z,... where x is the quantum length for the highest priority queue, y the next highest, and so forth |
| -A ALLOTMENTLIST | length of time allotment per queue level, specified as x,y,z,... where x is the # of time slices for the highest priority queue, y the next highest, and so forth |
| -j NUMJOBS | number of jobs in the system |
| -m MAXLEN | max run-time of a job (if randomly generating) |
| -M MAXIO | max I/O frequency of a job (if randomly generating) |
| -B BOOST | how often to boost the priority of all jobs back to high priority |
| -i IOTIME | how long an I/O should last (fixed constant) |
| -S | reset and stay at same priority level when issuing I/O |
| -I | if specified, jobs that finished I/O move immediately to front of current queue |
| -l JLIST | a comma-separated list of jobs to run, in the form x1,y1,z1:x2,y2,z2:... where x is start time, y is run time, and z is how often the job issues an I/O request |
| -c | compute answers for me |

./mlfq.py -n 2 -j 2 -m 10 -q 5 -s 30 -c

Execution Trace:

* Time slice, nachdem zum nächsten Job gewechselt wird und Prioritätsabstufung
* Restliche länge vom Job
* I/O start unterbricht time slice von Job 0
* I/O braucht 5 ticks

[ time 0 ] JOB BEGINS by JOB 0

[ time 0 ] JOB BEGINS by JOB 1

[ time 0 ] Run JOB 0 at PRIORITY 1 [ TICKS 4 ALLOT 1 TIME 4 (of 5) ]

[ time 1 ] Run JOB 0 at PRIORITY 1 [ TICKS 3 ALLOT 1 TIME 3 (of 5) ]

[ time 2 ] Run JOB 0 at PRIORITY 1 [ TICKS 2 ALLOT 1 TIME 2 (of 5) ]

[ time 3 ] IO\_START by JOB 0

[ time 3 ] Run JOB 1 at PRIORITY 1 [ TICKS 4 ALLOT 1 TIME 0 (of 1) ]

[ time 4 ] FINISHED JOB 1

[ time 4 ] IDLE

[ time 5 ] IDLE

[ time 6 ] IDLE

[ time 7 ] IDLE

[ time 8 ] IO\_DONE by JOB 0

[ time 8 ] Run JOB 0 at PRIORITY 1 [ TICKS 1 ALLOT 1 TIME 1 (of 5) ]

[ time 9 ] Run JOB 0 at PRIORITY 1 [ TICKS 0 ALLOT 1 TIME 0 (of 5) ]

[ time 10 ] FINISHED JOB 0

1. Run a few randomly-generated problems with just two jobs and two queues; compute the MLFQ execution trace for each. Make your life easier by limiting the length of each job and turning off I/Os.
2. How would you run the scheduler to reproduce each of the examples in the chapter?

* ./mlfq.py -n 3 -j 1 -m 100 -M 0 -c -> 3 ques, 1 job max length 100, keine I/O (8.2)
* ./mlfq.py -n 3 -l 0,180,0:100,20,0 -M 0 -c (8.3 links)
* ./mlfq.py -n 3 -q 1 -a 10 -S -l 0,175,0:50,25,1 -c (8.3 rechts)
* ./mlfq.py -n 3 -S -l 0,175,0:50,25,5:50,25,5 -c (8.4 links)
* (8.4 rechts)
* (8.5 links)
* (8.5 rechts)
* ./mlfq.py -n 3 -S -l 0,175,0:50,25,5:50,25,5 -c (8.6)

1. How would you configure the scheduler parameters to behave just like a round-robin scheduler?

./mlfq.py -n 1 -M 0

* Es gibt nur eine Queue und kein I/O

1. Craft a workload with two jobs and scheduler parameters so that one job takes advantage of the older Rules 4a and 4b (turned on with the -S flag) to game the scheduler and obtain 99% of the CPU over a particular time interval.

./mlfq.py -n 2 -q 100 -a 1 -i 1 -S -l 0,500,0:0,500,99 -c

Startzeit, job länge, wann io ausgelöst werden soll, timeslice

1. Given a system with a quantum length of 10 ms in its highest queue, how often would you have to boost jobs back to the highest priority level (with the -B flag) in order to guarantee that a single longrunning (and potentially-starving) job gets at least 5% of the CPU?

./mlfq.py -n 2 -q 20 -a 1 -i 1 -S -l 0,200,0:0,200,19 -c -B 200

* Bei einer quantum length (time slice) von 10ms muss mindedestens alle 200ms um mindestens eine cpu last von 5% zu bekommen. Allerdings ist es auch abhängig davon wie viele jobs in der gleichen queue sind

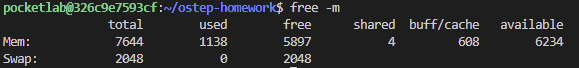
1. One question that arises in scheduling is which end of a queue to add a job that just finished I/O; the -I flag changes this behavior for this scheduling simulator. Play around with some workloads and see if you can see the effect of this flag.

./mlfq.py -j 2 -q 5 -l 0,30,3:0,30,5

* Sobald das I/O des Jobs fertig ist, geht’s mit ihm weiter bis sein allotment zuende ist und der andere Prozess, der vielleicht während des I/Os bearbeitet wurde, wird wieder pausiert.

# Kapitel 13 – Abstraction: Address Space

**Questions**

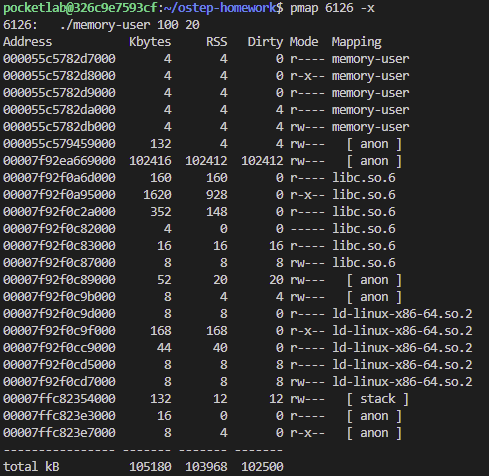
1. The first Linux tool you should check out is the very simple tool free. First, type man free and read its entire manual page; it’s short, don’t worry!
   * + free displays the total amount of free and used physical and swap memory in the system, as well as the buffers and caches used by the kernel
2. Now, run free, perhaps using some of the arguments that might be useful (e.g., -m, to display memory totals in megabytes). How much memory is in your system? How much is free? Do these numbers match your intuition?

shared -> systbibl. / memory-map | buff/cache -> daten von I/O zwischengespeichert

1. Next, create a little program that uses a certain amount of memory, called memory-user.c. This program should take one commandline argument: the number of megabytes of memory it will use. When run, it should allocate an array, and constantly stream through the array, touching each entry. The program should do this indefinitely, or, perhaps, for a certain amount of time also specified at the command line.

* Siehe memory-user.c

1. Now, while running your memory-user program, also (in a different terminal window, but on the same machine) run the free tool. How do the memory usage totals change when your program is running? How about when you kill the memory-user program? Do the numbers match your expectations? Try this for different amounts of memory usage. What happens when you use really large amounts of memory?

* Verhält sich wie erwartet

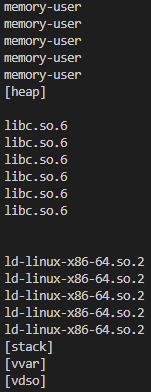
1. Let’s try one more tool, known as pmap. Spend some time, and read the pmap manual page in detail.

* The pmap reports the memory map of a process or processes.

Heap: ist Zeile 5-7 (da wo der Mode rw hat)

1. To use pmap, you have to know the process ID of the process you’re interested in. Thus, first run **ps auxw** to see a list of all processes; then, pick an interesting one, such as a browser. You can also use your memory-user program in this case (indeed, you can even have that program call getpid() and print out its PID for your convenience).

* pmap [options] [pid] -> (pids durch den befehl ps anzeigen)

1. Now run pmap on some of these processes, using various flags (like -X) to reveal many details about the process. What do you see? How many different entities make up a modern address space, as opposed to our simple conception of code/stack/heap?
2. Finally, let’s run pmap on your memory-user program, with different amounts of used memory. What do you see here? Does the output from pmap match your expectations?

# Kapitel 14 – Interlude: Memory API

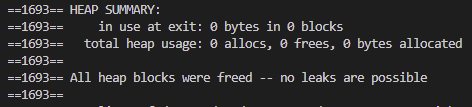
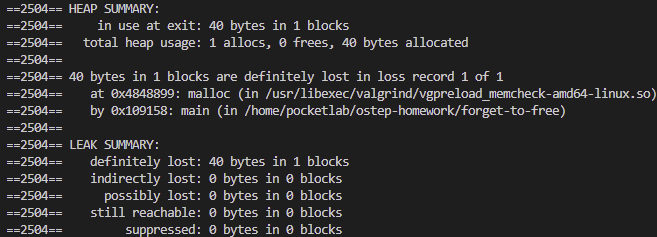
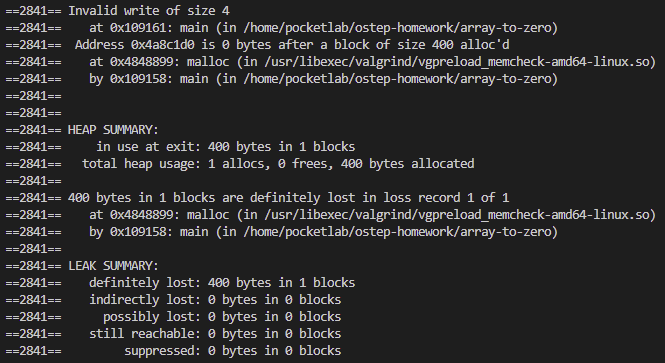
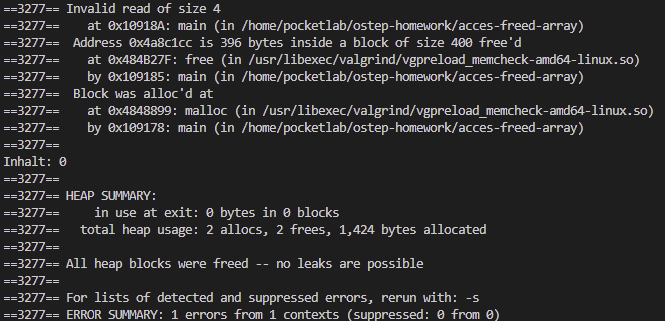
Questions

1. First, write a simple program called null.c that creates a pointer to an integer, sets it to NULL, and then tries to dereference it. Compile this into an executable called null. What happens when you run this program?

* Das Programm gibt ein Segmentation fault aus, weil es versucht auf ungültige Speicheradressen zuzugreifen

1. Next, compile this program with symbol information included (with the -g flag). Run the program under the debugger by typing gdb null and then, once gdb is running, typing run. What does gdb show you?

* Das -g-Flag sorgt dafür, dass das Kompilat mehr Informationen wie Variablennamen und Zeilennummern enthält.
* Gdb ist ein detailierterer Debugger

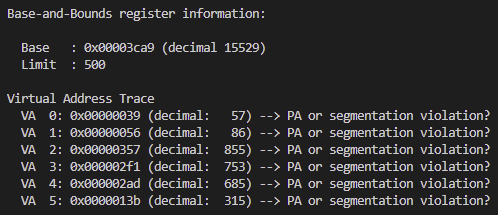
1. Finally, use the valgrind tool on this program. We’ll use the memcheck tool that is a part of valgrind to analyze what happens. Run this by typing in the following: valgrind --leak-check=yes null. What happens when you run this? Can you interpret the output from the tool?
2. Write a simple program that allocates memory using malloc() but forgets to free it before exiting. What happens when this program runs? Can you use gdb to find any problems with it? How about valgrind (again with the --leak-check=yes flag)?
3. Write a program that creates an array of integers called data of size 100 using malloc; then, set data[100] to zero. What happens when you run this program? What happens when you run this program using valgrind? Is the program correct?
4. Create a program that allocates an array of integers (as above), frees them, and then tries to print the value of one of the elements of the array. Does the program run? What happens when you use valgrind on it?
5. Now pass a funny value to free (e.g., a pointer in the middle of the array you allocated above). What happens? Do you need tools to find this type of problem?

* Vielleicht nicht möglich, da ja der allocierte bereich nur durch einen start und end pointer gekennzeichnet ist

1. Try out some of the other interfaces to memory allocation. For example, create a simple vector-like data structure and related routines that use realloc() to manage the vector. Use an array to store the vectors elements; when a user adds an entry to the vector, use realloc() to allocate more space for it. How well does such a vector perform? How does it compare to a linked list? Use valgrind to help you find bugs.
2. Spend more time and read about using gdb and valgrind. Knowing your tools is critical; spend the time and learn how to become an expert debugger in the UNIX and C environment.

# Kapitel 15 – Address Translation

./relocation.py



|  |  |
| --- | --- |
| -s SEED | the random seed |
| -a ASIZE | address space size (e.g., 16, 64k, 32m, 1g) |
| -p PSIZE | physical memory size (e.g., 16, 64k, 32m, 1g) |
| -n NUM | number of virtual addresses to generate |
| -b BASE | value of base register |
| -l LIMIT | value of limit register |
| -c | compute answers for me |



Questions

1. Run with seeds 1, 2, and 3, and compute whether each virtual address generated by the process is in or out of bounds. If in bounds, compute the translation.

* Sobald die dezimale Adresse über dem Limit ist, ist es eine SEGMENTATION VIOLATION

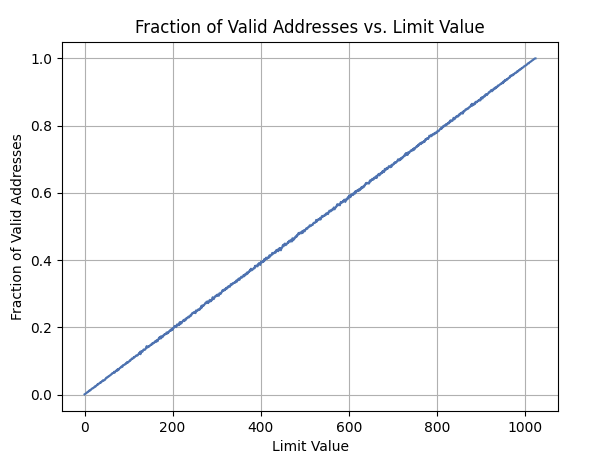
1. Run with these flags: -s 0 -n 10. What value do you have set -l (the bounds register) to in order to ensure that all the generated virtual addresses are within bounds?

* Die Größe der höchsten Adresse plus 1 (hier -l 929 +1)

1. Run with these flags: -s 1 -n 10 -l 100. What is the maximum value that base can be set to, such that the address space still fits into physical memory in its entirety?

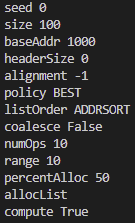
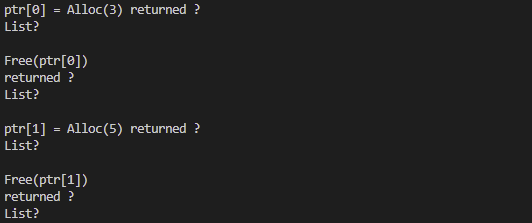
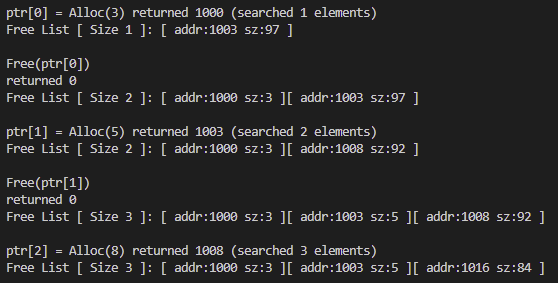
* Die Base kann maximal 16284 haben, da sonst der phyische Speicher nicht reicht

1. Run some of the same problems above, but with larger address spaces (-a) and physical memories (-p).



1. What fraction of randomly-generated virtual addresses are valid, as a function of the value of the bounds register? Make a graph from running with different random seeds, with limit values ranging from 0 up to the maximum size of the address space.

# Kapitel 17 – Free-Space Management

7./malloc.py,



|  |  |
| --- | --- |
| -s SEED | the random seed |
| -S HEAPSIZE | size of the heap |
| -b BASEADDR | base address of heap |
| -H HEADERSIZE | size of the header |
| -a ALIGNMENT | align allocated units to size; -1->no align |
| -p POLICY | list search (BEST, WORST, FIRST) |
| -l ORDER | list order (ADDRSORT, SIZESORT+, SIZESORT-, INSERT-FRONT, INSERT-BACK) |
| -C | coalesce the free list? |
| -n OPSRANGE | max alloc size |
| -P OPSPALLOC | percent of ops that are allocs |
| -A OPSLIST | instead of random, list of ops (+10,-0,etc) |
| -c | compute answers for me |



**Questions**



1. First run with the flags -n 10 -H 0 -p BEST -s 0 to generate a few random allocations and frees. Can you predict what alloc()/free() will return? Can you guess the state of the free list after each request? What do you notice about the free list over time?



* Die free list wird nicht länger, da durch die Best Fit Methode passende freie Block-Fragmente wieder genutzt werden können

1. How are the results different when using a WORST fit policy to search the free list (-p WORST)? What changes?

* Die free list wird beim Freigeben länger, da immer vom größten freien Block ein neues Fragment genommen wird

1. What about when using FIRST fit (-p FIRST)? What speeds up when you use first fit?

* Die Anazhl an durchsuchten Elementen wird kleiner, da nicht mehr die ganze free list durchsucht wird, sondern der erst beste genommen wird

1. For the above questions, how the list is kept ordered can affect the time it takes to find a free location for some of the policies. Use the different free list orderings (-l ADDRSORT, -l SIZESORT+, -l SIZESORT-) to see how the policies and the list orderings interact.

* **ADDRSORT** erleichtert das Coalescing und hilft bei Strategien, die auf eine einfache Blockverwaltung setzen.
* **SIZESORT+** ist ideal für Best Fit, um die zuerst kommenden kleineren Fragmente effizient zu nutzen.
* **SIZESORT-** ist optimal für Worst Fit, da die größten Blöcke zuerst kommen und für größere Anforderungen bewahrt werden.

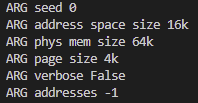
1. Coalescing of a free list can be quite important. Increase the number of random allocations (say to -n 1000). What happens to larger allocation requests over time? Run with and without coalescing (i.e., without and with the -C flag). What differences in outcome do you see? How big is the free list over time in each case? Does the ordering of the list matter in this case?

* Mit -C flag = free list kurz (max 5) , ohne -C flag = free list sehr lang (~30-40)
* Sortierung macht viel aus -C flag funktioniert nur bei ADDRSORT effektiv

1. What happens when you change the percent allocated fraction -P to higher than 50? What happens to allocations as it nears 100? What about as the percent nears 0?

* Je höher die Wahrscheinlichkeit, desto weniger wird wieder freigegeben, weniger fragmente und die größe des freien Speichers wird weniger
* Je kleiner, desto mehr frees und einzelne Fragemente ohne -C flag, Fragmente können wieder genutzt werden

1. What kind of specific requests can you make to generate a highlyfragmented free space? Use the -A flag to create fragmented free lists, and see how different policies and options change the organization of the free list.



# Kapitel 18 – Paging: Introduction

16KB Address Space, 4KB pages -> 16/4 = 4 VPNs

Frage: VA 0x00003229 (decimal: 12841) --> PA or invalid address?

Page Table:

0x8000000c <-- PTBR

0x00000000

0x00000000

0x80000006 <-- PTE für VPN=3

VA 0x00003229

VPN -> what to index into page table with

Offset

16KB VA -> 14 bits Virtuel Address

4KB Page -> 2 bits VPN, 12 bits offset

VA 0x3229 = 11 001000101001

VPN offset

Address of PTE(VPN = 3): PTBR + (3 \* sizeof(PTE))

PTE: 0x80000006 -> 1000 0000 0000 0000 0000 0000 0000 0110

1 is high bit -> VALID

PFN -> 0x006

VPN: 0x3 offset: 0x229

PFN: 0x6 offset: 0x229

In Binär: 0110 001000101001

-> 0x6229

|  |  |
| --- | --- |
| -h | show this help message and exit |
| -s SEED | the random seed |
| -a ASIZE | address space size (e.g., 16, 64k, ...) |
| -p PSIZE | physical memory size (e.g., 16, 64k, ...) |
| -P PAGESIZE | page size (e.g., 4k, 8k, ...) |
| -n NUM | number of virtual addresses to generate |
| -u USED | percent of address space that is used |
| -v | verbose mode |
| -c | compute answers for me |

Schritte:

1. Virtuelle Adresse in VPN und Offset umrechnen

2. Mit VPN im Pagetable nach PTE suchen (anfangend bei 0)

3. 1. bit der PTE ist Valid-bit, alle andere ist PFN

4. Physikalische Adresse: PFN + Offset -> ganze Binär in Hex umrechnen

Schnell ablesen:

dezimalwert der VA geteilt durch 4046 (page size) -> gerundet VPN

restlicher teil des PTE ist dann PFN + den retst von der VA

Questions

1. Before doing any translations, let’s use the simulator to study how linear page tables change size given different parameters. Compute the size of linear page tables as different parameters change. Some suggested inputs are below; by using the -v flag, you can see how many page-table entries are filled. First, to understand how linear page table size changes as the address space grows, run with these flags:

-P 1k -a 1m -p 512m -v -n 0 -> 1024 PTE

-P 1k -a 2m -p 512m -v -n 0 -> 2048 PTE

-P 1k -a 4m -p 512m -v -n 0 -> 4096 PTE

Then, to understand how linear page table size changes as page size grows:

-P 1k -a 1m -p 512m -v -n 0 -> 1024 PTE

-P 2k -a 1m -p 512m -v -n 0 -> 512 PTE

-P 4k -a 1m -p 512m -v -n 0 -> 256 PTE

Before running any of these, try to think about the expected trends. How should page-table size change as the address space grows? As the page size grows? Why not use big pages in general?

* + Je größer der Adressbereich, desto mehr PTE
  1. Now let’s do some translations. Start with some small examples, and change the number of pages that are allocated to the address space with the -u flag. For example:

-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

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

* 1. Now let’s try some different random seeds, and some different (and sometimes quite crazy) address-space parameters, for variety:

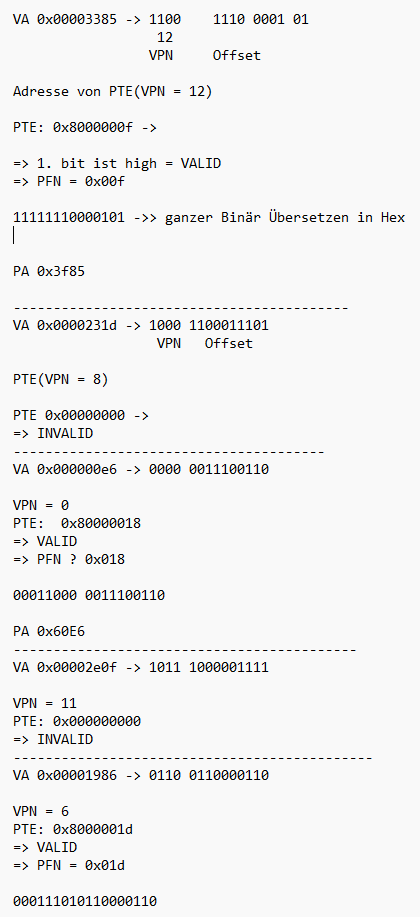
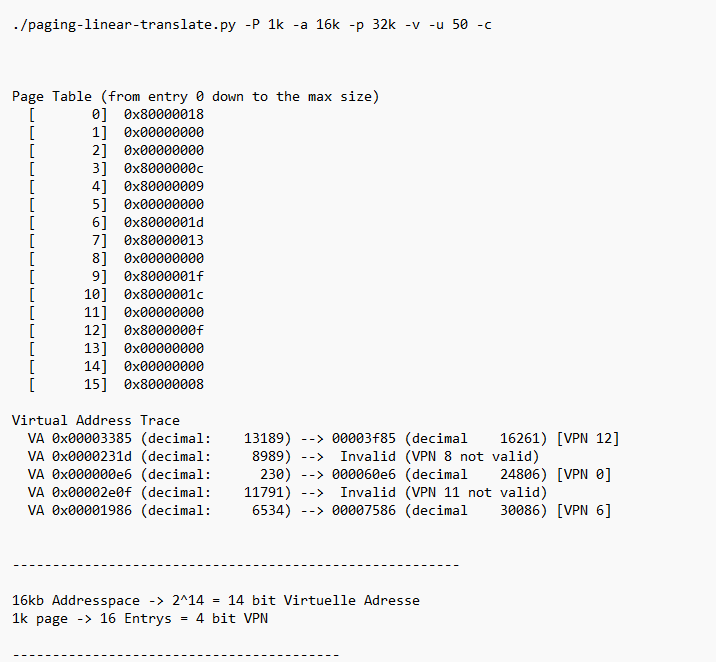
-P 8 -a 32 -p 1024 -v -s 1

-P 8k -a 32k -p 1m -v -s 2

-P 1m -a 256m -p 512m -v -s 3

Which of these parameter combinations are unrealistic? Why?

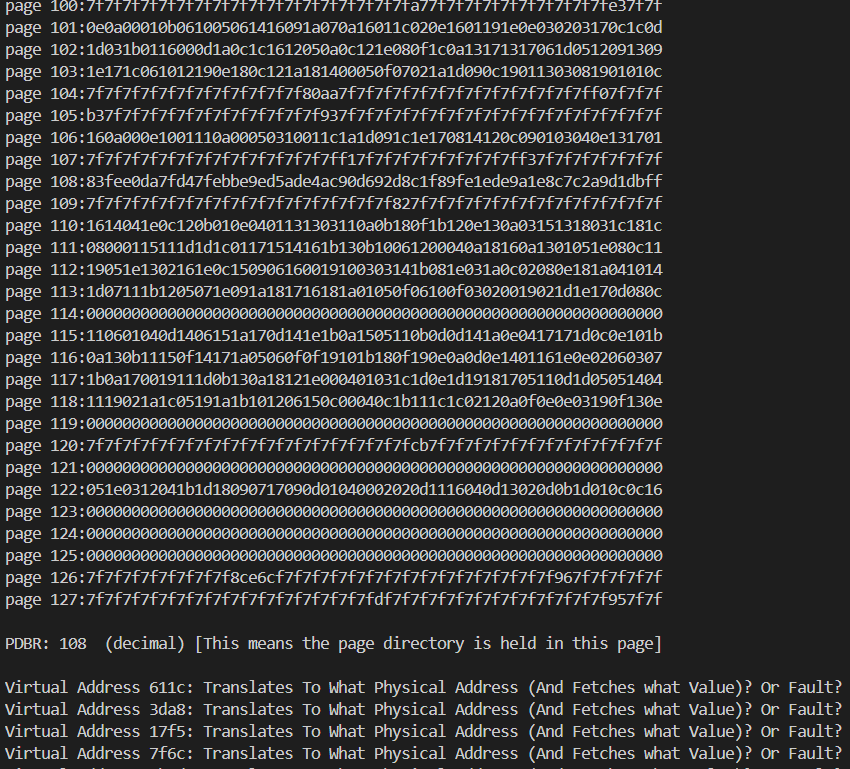
* 1. Use the program to try out some other problems. Can you find the limits of where the program doesn’t work anymore? For example, what happens if the address-space size is bigger than physical memory?



# Kapitel 20 – Paing: Smaller Tables

Schritte:

1. Virtuelle Adresse in PDE, PTE und Offset übersetzen

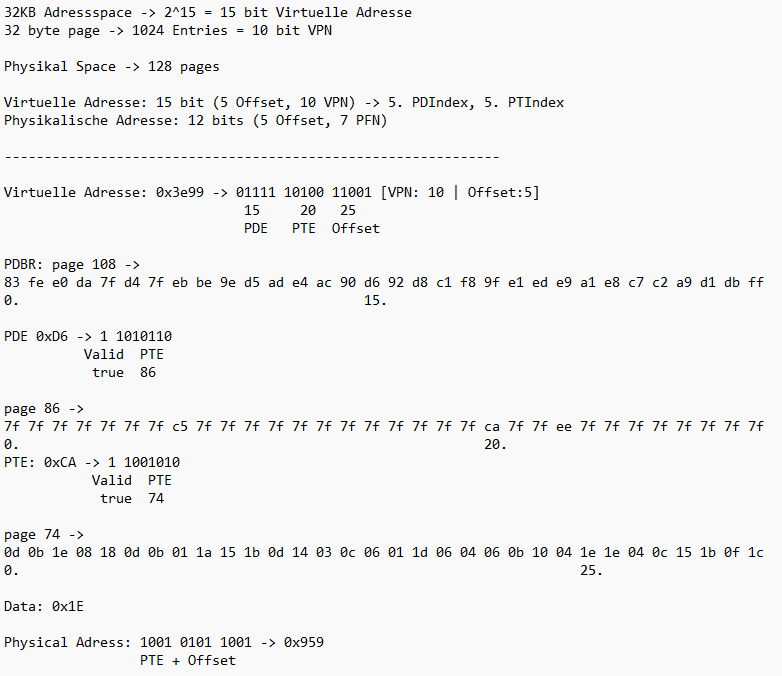
2. im PDBR nach der Stelle des PDEs (beginnend mit 0.) suchen

3. Page mit der Zahl (aus der PDE-Stelle) der PDBR nehmen und darin nach der Stelle des PTE suchen

4. Page mit der Zahl (aus der PTE-Stelle) der PDE nehmen und darin nach der Stelle des Offsets nach der Data suchen

5. Physikalische Adresse besteht Page Zahl des PTE und dem Offset

Questions

1. With a linear page table, you need a single register to locate the page table, assuming that hardware does the lookup upon a TLB miss. How many registers do you need to locate a two-level page table? A three-level table?



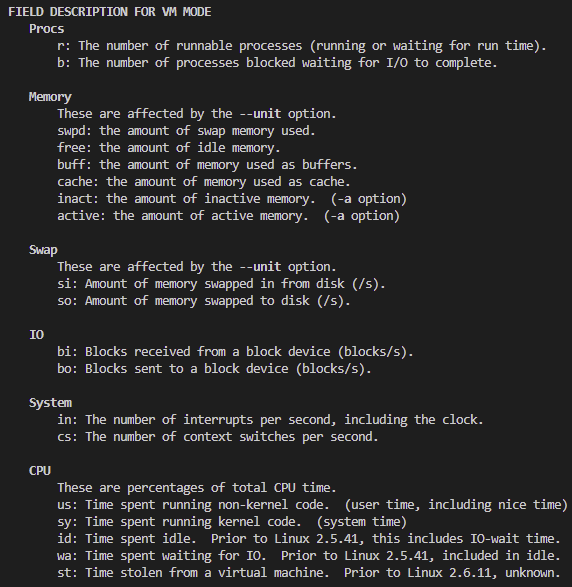
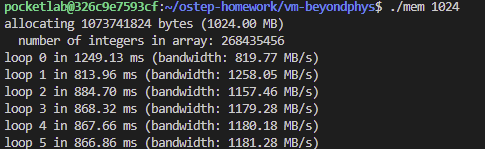
1. Use the simulator to perform translations given random seeds 0, 1, and 2, and check your answers using the -c flag. How many memory references are needed to perform each lookup?

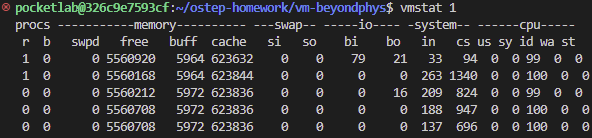


1. Given your understanding of how cache memory works, how do you think memory references to the page table will behave in the cache? Will they lead to lots of cache hits (and thus fast accesses?) Or lots of misses (and thus slow accesses)?



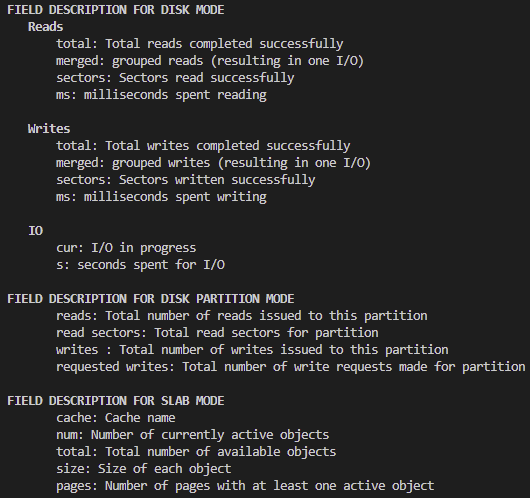
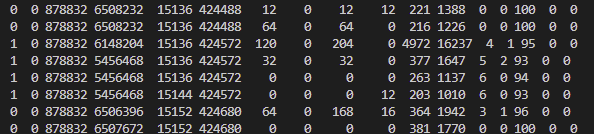
# Kapitel 21 - Beyond Physical Memory: Mechanisms

./mem.c 1024

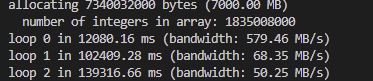
./vmstat

KiB

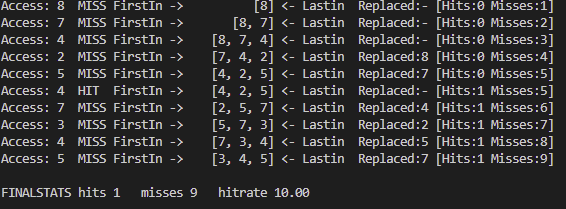
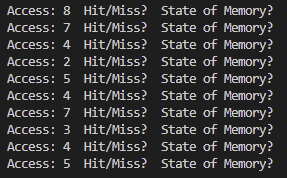
**Questions**

1. First, open two separate terminal connections to the same machine, so that you can easily run something in one window and the other. Now, in one window, run vmstat 1, which shows statistics about machine usage every second. Read the man page, the associated README, and any other information you need so that you can understand its output. Leave this window running vmstat for the rest of the exercises below. Now, we will run the program mem.c but with very little memory usage. This can be accomplished by typing ./mem 1 (which uses only 1 MB of memory). How do the CPU usage statistics change when running mem? Do the numbers in the user time column make sense? How does this change when running more than one instance of mem at once?
2. Let’s now start looking at some of the memory statistics while running mem. We’ll focus on two columns: swpd (the amount of virtual memory used) and free (the amount of idle memory). Run ./mem 1024 (which allocates 1024 MB) and watch how these values change. Then kill the running program (by typing control-c) and watch again how the values change. What do you notice about the values? In particular, how does the free column change when the program exits? Does the amount of free memory increase by the expected amount when mem exits?
   * Ja sinkt und steigt um ungefähr 1024 mb



1. We’ll next look at the swap columns (si and so), which indicate how much swapping is taking place to and from the disk. Of course, to activate these, you’ll need to run mem with large amounts of memory. First, examine how much free memory is on your Linux system (for example, by typing cat /proc/meminfo; type man proc for details on the /proc file system and the types of information you can find there). One of the first entries in /proc/meminfo is the total amount of memory in your system. Let’s assume it’s something like 8 GB of memory; if so, start by running mem 4000 (about 4 GB) and watching the swap in/out columns. Do they ever give non-zero values? Then, try with 5000, 6000, etc. What happens to these values as the program enters the second loop (and beyond), as compared to the first loop? How much data (total) are swapped in and out during the second, third, and subsequent loops? (do the numbers make sense?)
   * Es wird zuerst viel ausgelagert (hier: 350 MiB) und 23 MiB in den Speicher geholt. Der Swap ist auch dementsprechend um 300 MiB größer geworden. Danach bleibt die Rate der eingelagerten und ausgelagerten Daten gleich bei jw. 80-90 MiB.
   * Der free bleibt dauerhaft bei ungefähr 120 MiB
   * Der Prozess wird ständig zw. Run und blocked geschoben weil so viel auf die Disk gewartet wird
2. Do the same experiments as above, but now watch the other statistics (such as CPU utilization, and block I/O statistics). How do they change when mem is running?
   * Am anfang sind us und sy (Modus wechsel) dazugekommen und als der speicher voll gelaufen ist, ist das wa( waiting for i/o) von 0 auf meistens 5
3. Now let’s examine performance. Pick an input for mem that comfortably fits in memory (say 4000 if the amount of memory on the system is 8 GB). How long does loop 0 take (and subsequent loops 1, 2, etc.)? Now pick a size comfortably beyond the size of memory (say 12000 again assuming 8 GB of memory). How long do the loops take here? How do the bandwidth numbers compare? How different is performance when constantly swapping versus fitting everything comfortably in memory? Can you make a graph, with the size of memory used by mem on the x-axis, and the bandwidth of accessing said memory on the y-axis? Finally, how does the performance of the first loop compare to that of subsequent loops, for both the case where everything fits in memory and where it doesn’t?
4. Swap space isn’t infinite. You can use the tool swapon with the -s flag to see how much swap space is available. What happens if you try to run mem with increasingly large values, beyond what seems to be available in swap? At what point does the memory allocation fail?
   * Gibt kein platz, mem failt
5. Finally, if you’re advanced, you can configure your system to use different swap devices using swapon and swapoff. Read the man pages for details. If you have access to different hardware, see how the performance of swapping changes when swapping to a classic hard drive, a flash-based SSD, and even a RAID array. How much can swapping performance be improved via newer devices? How close can you get to in-memory performance?

# Kapitel 22 – Beyond Physical Memory: Policies

./paging-policy.py

|  |  |
| --- | --- |
| -a ADDRESSES | a set of comma-separated pages to access; -1 means randomly generate |
| -f ADDRESSFILE | a file with a bunch of addresses in it |
| -n NUMADDRS | if -a (--addresses) is -1, this is the number of addrs to generate |
| -p POLICY | replacement policy: FIFO, LRU, OPT(peaks into futur), UNOPT, RAND, CLOCK |
| -b CLOCKBITS | for CLOCK policy, how many clock bits to use |
| -C CACHESIZE | size of the page cache, in pages |
| -m MAXPAGE | if randomly generating page accesses, this is the max page number |
| -s SEED | random number seed |
| -N | do not print out a detailed trace |
| -c | compute answers for me |

Questions

1. Generate random addresses with the following arguments: -s 0 -n 10, -s 1 -n 10, and -s 2 -n 10. Change the policy from FIFO, to LRU, to OPT. Compute whether each access in said address traces are hits or misses.
2. For a cache of size 5, generate worst-case address reference streams for each of the following policies: FIFO, LRU, and MRU (worst-case reference streams cause the most misses possible. For the worst case reference streams, how much bigger of a cache is needed to improve performance dramatically and approach OPT?
   * ./paging-policy.py -C 3 -a 1,2,3,4,1,2,3,4 -p FIFO/LRU -c
   * ./paging-policy.py -C 3 -a 1,2,3,4,3,4,3 -p MRU -c
   * Der Index muss immer so groß sein, wie die Anzahl der unterschiedlichen Adresse um OPT zu erreichen
3. Generate a random trace (use python or perl). How would you expect the different policies to perform on such a trace?
   * ./paging-policy.py -C 3 -a -1 -n 20 -m 10 -s 1
   * **1. FIFO (First In, First Out):**
     + With a random trace, FIFO is unlikely to exploit any temporal locality, as it does not consider recent usage.Performance will likely be average, with frequent misses.
   * **2. LRU (Least Recently Used):**
     + LRU typically performs better than FIFO when there is temporal locality (e.g., recently used pages are accessed again). With random traces, temporal locality may not be present, so LRU's performance could be similar to FIFO.
   * **3. OPT (Optimal):**
     + OPT gives the best possible performance for a given trace, as it perfectly predicts future accesses.
   * **4. RAND (Random):**
     + Similar to FIFO with random traces, RAND does not exploit locality, and its performance is usually poor or average. However, in some cases, RAND may perform better than FIFO due to its unpredictability.
   * **5. CLOCK:**
     + CLOCK approximates LRU but with lower overhead. With random traces, its performance may align closely with FIFO or LRU.
   * **6. UNOPT (Suboptimal):**
     + UNOPT intentionally performs poorly by design and will have the highest number of misses on any trace, especially random ones.
4. Now generate a trace with some locality. How can you generate such a trace? How does LRU perform on it? How much better than RAND is LRU? How does CLOCK do? How about CLOCK with different numbers of clock bits?
   * ./paging-policy.py -C 3 -a 1,2,3,4,2,4,3,2,3
5. Use a program like valgrind to instrument a real application and generate a virtual page reference stream. For example, running valgrind --tool=lackey --trace-mem=yes ls will output a nearly-complete reference trace of every instruction and data reference made by the program ls. To make this useful for the simulator above, you’ll have to first transform each virtual memory reference into a virtual page-number reference (done by masking off the offset and shifting the resulting bits downward). How big of a cache is needed for your application trace in order to satisfy a large fraction of requests? Plot a graph of its working set as the size of the cache increases.

# Kapitel 26 - Concurrency: An Introduction

This program, x86.py, allows you to see how different thread interleavings either cause or avoid race conditions. See the README for details on how the program works, then answer the questions below.

|  |  |
| --- | --- |
| -s SEED | the random seed |
| -t NUMTHREADS | number of threads |
| -p PROGFILE | source program (in .s) |
| -i INTFREQ | interrupt frequency |
| -r | if interrupts are random |
| -a ARGV | comma-separated per-thread args (e.g., ax=1,ax=2 sets thread 0 ax reg to 1 and thread 1 ax reg to 2); specify multiple regs per thread via colon-separated list (e.g., ax=1:bx=2,cx=3 sets thread 0 ax and bx and just cx for thread 1) |
| -L LOADADDR | address where to load code |
| -m MEMSIZE | size of address space (KB) |
| -M MEMTRACE | comma-separated list of addrs to trace (e.g., 20000,20001) |
| -R REGTRACE | comma-separated list of regs to trace (e.g., ax,bx,cx,dx) |
| -C | should we trace condition codes |
| -S | print some extra stats |
| -v | print some extra info |
| -c | compute answers for me |

Questions

1. Let’s examine a simple program, “loop.s”. First, just read and understand it. Then, run it with these arguments (./x86.py -p loop.s -t 1 -i 100 -R dx) This specifies a single thread, an interrupt every 100 instructions, and tracing of register %dx. What will %dx be during the run? Use the -c flag to check your answers; the answers, on the left, show the value of the register (or memory value) after the instruction on the right has run.
2. Same code, different flags: (./x86.py -p loop.s -t 2 -i 100 -a dx=3,dx=3 -R dx) This specifies two threads, and initializes each %dx to 3. What values will %dx see? Run with -c to check. Does the presence of multiple threads affect your calculations? Is there a race in this code?
   * Ne weil die jeweiligen dx im Thread eigenen Stack/Registern gespeichert werden
3. Run this: ./x86.py -p loop.s -t 2 -i 3 -r -a dx=3,dx=3 -R dx This makes the interrupt interval small/random; use different seeds (-s) to see different interleavings. Does the interrupt frequency change anything?
   * Nope
4. Now, a different program, looping-race-nolock.s, which accesses a shared variable located at address 2000; we’ll call this variable value. Run it with a single thread to confirm your understanding: ./x86.py -p looping-race-nolock.s -t 1 -M 2000 What is value (i.e., at memory address 2000) throughout the run? Use -c to check.
   * 1
5. Run with multiple iterations/threads: ./x86.py -p looping-race-nolock.s -t 2 -a bx=3 -M 2000 Why does each thread loop three times? What is final value of value?
   * Weil bx 3 mal subtrahiert wird, um die jump bedingung nicht mehr zu erfüllen.
   * Value = 6
6. Run with random interrupt intervals: ./x86.py -p looping-race-nolock.s -t 2 -M 2000 -i 4 -r -s 0 with different seeds (-s 1, -s 2, etc.) Can you tell by looking at the thread interleaving what the final value of value will be? Does the timing of the interrupt matter? Where can it safely occur? Where not? In other words, where is the critical section exactly?
   * Je nach dem ob der erste Interrupt vor dem 2. move kommt, zieht sich der zweite Thread den alten oder neuen Wert von value.
   * Solange der interrupt nach dem zurückschreiben kommt, ist es kein Problem
7. Now examine fixed interrupt intervals: ./x86.py -p looping-race-nolock.s -a bx=1 -t 2 -M 2000 -i 1 What will the final value of the shared variable value be? What about when you change -i 2, -i 3, etc.? For which interrupt intervals does the program give the “correct” answer?
   * Für einen interval, der den kritischen abschnitt zusammen ausführen lässt (hier 3)
8. Run the same for more loops (e.g., set -a bx=100). What interrupt intervals (-i) lead to a correct outcome? Which intervals are surprising?
   * Entweder die länge des KA oder ein vielfaches der Gesamtanzahl der Instruktionen
9. One last program: wait-for-me.s. Run: ./x86.py -p wait-for-me.s -a ax=1,ax=0 -R ax -M 2000 This sets the %ax register to 1 for thread 0, and 0 for thread 1, and watches %ax and memory location 2000. How should the code behave? How is the value at location 2000 being used by the threads? What will its final value be?
   * Er wird als Conditional Variable verwendet und gibt signale zwischen den Threads weiter
10. Now switch the inputs: ./x86.py -p wait-for-me.s -a ax=0,ax=1 -R ax -M 2000 How do the threads behave? What is thread 0 doing? How would changing the interrupt interval (e.g., -i 1000, or perhaps to use random intervals) change the trace outcome? Is the program efficiently using the CPU?
    * Thread 0 wartet so lange darauf, bis durch ein Interrupt auf Thread 1 umgeschaltet wird und er dann das Signal gibt, aus der Schleife zu kommen.

# Kapitel 27 - Interlude: Thread API

In this section, we’ll write some simple multi-threaded programs and use a specific tool, called helgrind, to find problems in these programs. Read the README in the homework download for details on how to build the programs and run helgrind.

Questions

1. First build main-race.c. Examine the code so you can see the (hopefully obvious) data race in the code. Now run helgrind (by typing valgrind --tool=helgrind main-race) to see how it reports the race. Does it point to the right lines of code? What other information does it give to you?
2. What happens when you remove one of the offending lines of code? Now add a lock around one of the updates to the shared variable, and then around both. What does helgrind report in each of these cases?
   * Es informiert über die existens der Locks, zeigt aber trotzdem ein Error für Race-Conditions
3. Now let’s look at main-deadlock.c. Examine the code. This code has a problem known as deadlock (which we discuss in much more depth in a forthcoming chapter). Can you see what problem it might have?
4. Now run helgrind on this code. What does helgrind report?
5. Now run helgrind on main-deadlock-global.c. Examine the code; does it have the same problem that main-deadlock.c has? Should helgrind be reporting the same error? What does this tell you about tools like helgrind?
6. Let’s next look at main-signal.c. This code uses a variable (done) to signal that the child is done and that the parent can now continue. Why is this code inefficient? (what does the parent end up spending its time doing, particularly if the child thread takes a long time to complete?)
7. Now run helgrind on this program. What does it report? Is the code correct?
8. Now look at a slightly modified version of the code, which is found in main-signal-cv.c. This version uses a condition variable to do the signaling (and associated lock). Why is this code preferred to the previous version? Is it correctness, or performance, or both?
9. Once again run helgrind on main-signal-cv. Does it report any errors?

# Kapitel 28 – Locks

This program, x86.py, allows you to see how different thread interleavings either cause or avoid race conditions. See the README for details on how the program works and answer the questions below.

Questions

1. Examine flag.s. This code “implements” locking with a single memory flag. Can you understand the assembly?
2. When you run with the defaults, does flag.s work? Use the -M and -R flags to trace variables and registers (and turn on -c to see their values). Can you predict what value will end up in flag?
3. Change the value of the register %bx with the -a flag (e.g., -a bx=2,bx=2 if you are running just two threads). What does the code do? How does it change your answer for the question above?
4. Set bx to a high value for each thread, and then use the -i flag to generate different interrupt frequencies; what values lead to a bad outcomes? Which lead to good outcomes?
   * Bei interrupt <= 3 kommt es zu race conditions, ansonsten funktioniert die flag
5. Now let’s look at the program test-and-set.s. First, try to understand the code, which uses the xchg instruction to build a simple locking primitive. How is the lock acquire written? How about lock release?
6. Now run the code, changing the value of the interrupt interval (-i) again, and making sure to loop for a number of times. Does the code always work as expected? Does it sometimes lead to an inefficient use of the CPU? How could you quantify that?
   * Es sperrt auch bei niedrigen interrupt frequenzen, thread spin-waits bis das lock wieder frei ist
7. Use the -P flag to generate specific tests of the locking code. For example, run a schedule that grabs the lock in the first thread, but then tries to acquire it in the second. Does the right thing happen? What else should you test?
8. Now let’s look at the code in peterson.s, which implements Peterson’s algorithm (mentioned in a sidebar in the text). Study the code and see if you can make sense of it.
9. Now run the code with different values of -i. What kinds of different behavior do you see? Make sure to set the thread IDs appropriately (using -a bx=0,bx=1 for example) as the code assumes it.
10. Can you control the scheduling (with the -P flag) to “prove” that the code works? What are the different cases you should show hold? Think about mutual exclusion and deadlock avoidance.
11. Now study the code for the ticket lock in ticket.s. Does it match the code in the chapter? Then run with the following flags: -a bx=1000,bx=1000 (causing each thread to loop through the critical section 1000 times). Watch what happens; do the threads spend much time spin-waiting for the lock?
12. How does the code behave as you add more threads?
13. Now examine yield.s, in which a yield instruction enables one thread to yield control of the CPU (realistically, this would be an OS primitive, but for the simplicity, we assume an instruction does the task). Find a scenario where test-and-set.s wastes cycles spinning, but yield.s does not. How many instructions are saved? In what scenarios do these savings arise?
14. Finally, examine test-and-test-and-set.s. What does this lock do? What kind of savings does it introduce as compared to test-and-set.s?

# Kapitel 29 - Lock-based Concurrent Data Structures

In this homework, you’ll gain some experience with writing concurrent code and measuring its performance. Learning to build code that performs well is a critical skill and thus gaining a little experience here with it is quite worthwhile.

Questions

1. We’ll start by redoing the measurements within this chapter. Use the call gettimeofday() to measure time within your program. How accurate is this timer? What is the smallest interval it can measure? Gain confidence in its workings, as we will need it in all subsequent questions. You can also look into other timers, such as the cycle counter available on x86 via the rdtsc instruction.
2. Now, build a simple concurrent counter and measure how long it takes to increment the counter many times as the number of threads increases. How many CPUs are available on the system you are using? Does this number impact your measurements at all?
3. Next, build a version of the sloppy counter. Once again, measure its performance as the number of threads varies, as well as the threshold. Do the numbers match what you see in the chapter?
4. Build a version of a linked list that uses hand-over-hand locking [MS04], as cited in the chapter. You should read the paper first to understand how it works, and then implement it. Measure its performance. When does a hand-over-hand list work better than a standard list as shown in the chapter?
5. Pick your favorite data structure, such as a B-tree or other slightly more interesting structure. Implement it, and start with a simple locking strategy such as a single lock. Measure its performance as the number of concurrent threads increases.
6. Finally, think of a more interesting locking strategy for this favorite data structure of yours. Implement it, and measure its performance. How does it compare to the straightforward locking approach?

# Kapitel 30 - Condition Variables

This homework lets you explore some real code that uses locks and condition variables to implement various forms of the producer/consumer queue discussed in the chapter. You’ll look at the real code, run it in various configurations, and use it to learn about what works and what doesn’t, as well as other intricacies. Read the README for details.

Questions

1. Our first question focuses on main-two-cvs-while.c (the working solution). First, study the code. Do you think you have an understanding of what should happen when you run the program?
2. Run with one producer and one consumer, and have the producer produce a few values. Start with a buffer (size 1), and then increase it. How does the behavior of the code change with larger buffers? (or does it?) What would you predict num full to be with different buffer sizes (e.g., -m 10) and different numbers of produced items (e.g., -l 100), when you change the consumer sleep string from default (no sleep) to -C 0,0,0,0,0,0,1?
   * Egal wie groß der Buffer ist, der Consumer-Counter ist immer so groß wie l
3. If possible, run the code on different systems (e.g., a Mac and Linux). Do you see different behavior across these systems?
   * Keine Ahnung
4. Let’s look at some timings. How long do you think the following execution, with one producer, three consumers, a single-entry shared buffer, and each consumer pausing at point c3 for a second, will take? ./main-two-cvs-while -p 1 -c 3 -m 1 -C 0,0,0,1,0,0,0:0,0,0,1,0,0,0:0,0,0,1,0,0,0 -l 10 -v -t
   * Es wird 12. C3 aufgerufen, daher wird insgesamt 12sek gewartet
   * Es geht nur der Counter vom ersten Consumer hoch, da für den 2. Und 3. Der Buffer leer ist
5. Now change the size of the shared buffer to 3 (-m 3). Will this make any difference in the total time?
   * Ja 11 aber versteh nicht warum, weil ja gleich oft der num\_fill 0 ist da es nicht mehr producer gibt
6. Now change the location of the sleep to c6 (this models a consumer taking something off the queue and then doing something with it), again using a single-entry buffer. What time do you predict in this case? ./main-two-cvs-while -p 1 -c 3 -m 1 -C 0,0,0,0,0,0,1:0,0,0,0,0,0,1:0,0,0,0,0,0,1 -l 10 -v -t
   * Bei c6 macht es kein unterschied weil danach der thread meistens schlafen geht
7. Finally, change the buffer size to 3 again (-m 3). What time do you predict now?
   * Macht kein unterschied von der länge
8. Now let’s look at main-one-cv-while.c. Can you configure a sleep string, assuming a single producer, one consumer, and a buffer of size 1, to cause a problem with this code?
   * Hier funktioniert alles weil es nur 1 consumer und 1 producer gibt
9. Now change the number of consumers to two. Can you construct sleep strings for the producer and the consumers so as to cause a problem in the code?
   * Es gibt fehler, da es mehrere Consumer gibt und er ausversehen den anderen Consumer anstelle vom Producer aufweckt
10. Now examine main-two-cvs-if.c. Can you cause a problem to happen in this code? Again consider the case where there is only one consumer, and then the case where there is more than one.
    * Bei if ist das Problem dass nur einmal gecheckt wird ob die bedingung noch besteht
11. Finally, examine main-two-cvs-while-extra-unlock.c. What problem arises when you release the lock before doing a put or a get? Can you reliably cause such a problem to happen, given the sleep strings? What bad thing can happen?
    * Die do\_fill methode ist nicht mehr geschützt und kann eine race-condition auslösen

# Kapitel 31 – Semaphores

In this homework, we’ll use semaphores to solve some well-known concurrency problems. Many of these are taken from Downey’s excellent “Little Book of Semaphores”3 , which does a good job of pulling together a number of classic problems as well as introducing a few new variants; interested readers should check out the Little Book for more fun. Each of the following questions provides a code skeleton; your job is to fill in the code to make it work given semaphores. On Linux, you will be using native semaphores; on a Mac (where there is no semaphore support), you’ll have to first build an implementation (using locks and condition variables, as described in the chapter). Good luck!

Questions

1. The first problem is just to implement and test a solution to the fork/join problem, as described in the text. Even though this solution is described in the text, the act of typing it in on your own is worthwhile; even Bach would rewrite Vivaldi, allowing one soon-to-be master to learn from an existing one. See fork-join.c for details. Add the call sleep(1) to the child to ensure it is working.
2. Let’s now generalize this a bit by investigating the rendezvous problem. The problem is as follows: you have two threads, each of which are about to enter the rendezvous point in the

code. Neither should exit this part of the code before the other enters it. Consider using two semaphores for this task, and see rendezvous.c for details.

1. Now go one step further by implementing a general solution to barrier synchronization. Assume there are two points in a sequential piece of code, called P1 and P2. Putting a barrier between P1 and P2 guarantees that all threads will execute P1 before any one thread executes P2. Your task: write the code to implement a barrier() function that can be used in this manner. It is safe to assume you know N (the total number of threads in the running program) and that all N threads will try to enter the barrier. Again, you should likely use two semaphores to achieve the solution, and some other integers to count things. See barrier.c for details.
2. Now let’s solve the reader-writer problem, also as described in the text. In this first take, don’t worry about starvation. See the code in reader-writer.c for details. Add sleep() calls to your code to demonstrate it works as you expect. Can you show the existence of the starvation problem?
3. Let’s look at the reader-writer problem again, but this time, worry about starvation. How can you ensure that all readers and writers eventually make progress? See reader-writer-nostarve.c for details.
4. Use semaphores to build a no-starve mutex, in which any thread that tries to acquire the mutex will eventually obtain it. See the code in mutex-nostarve.c for more information.
5. Liked these problems? See Downey’s free text for more just like them. And don’t forget, have fun! But, you always do when you write code, no?

# Kapitel 32 – Common Concurrency Problems

This homework lets you explore some real code that deadlocks (or avoids deadlock). The different versions of code correspond to different approaches to avoiding deadlock in a simplified vector add() routine. See the README for details on these programs and their common substrate.

Questions

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 -n 2 -l 1 -v, which instantiates two threads (-n 2), each of which does one vector add (-l 1), and does so in verbose mode (-v). Make sure you understand the output. How does the output change from run to run?
2. Now add the -d flag, and change the number of loops (-l) from 1 to higher numbers. What happens? Does the code (always) deadlock?
3. 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?
4. 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? Also, why is there a special case in this vector add() routine when the source and destination vectors are the same?
5. 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?
6. What happens if you turn on the parallelism flag (-p)? How much would you expect performance to change when each thread is working on adding different vectors (which is what -p enables) versus working on the same ones?
7. 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?
8. 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, when running both with -p and without it?
9. 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?
10. Now compare its performance to the other versions, 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?

# Kapitel 37 – File Disks

This homework uses disk.py to familiarize you with how a modern hard drive works. It has a lot of different options, and unlike most of the other simulations, has a graphical animator to show you exactly what happens when the disk is in action. See the README for details.

1. Compute the seek, rotation, and transfer times for the following sets of requests: -a 0, -a 6, -a 30, -a 7,30,8, and finally -a 10,11,12,13.

2. Do the same requests above, but change the seek rate to different values: -S 2, -S 4, -S 8, -S 10, -S 40, -S 0.1. How do the times change?

3. Do the same requests above, but change the rotation rate: -R 0.1, -R 0.5, -R 0.01. How do the times change?

4. FIFO is not always best, e.g., with the request stream -a 7,30,8, what order should the requests be processed in? Run the shortest seek-time first (SSTF) scheduler (-p SSTF) on this workload; how long should it take (seek, rotation, transfer) for each request to be served?

5. Now use the shortest access-time first (SATF) scheduler (-p SATF). Does it make any difference for -a 7,30,8 workload? Find a set of requests where SATF outperforms SSTF; more generally, when is SATF better than SSTF?

6. Here is a request stream to try: -a 10,11,12,13. What goes poorly when it runs? Try adding track skew to address this problem (-o skew). Given the default seek rate, what should the skew be to maximize performance? What about for different seek rates (e.g., -S 2, -S 4)? In general, could you write a formula to figure out the skew?

7. Specify a disk with different density per zone, e.g., -z 10,20,30, which specifies the angular difference between blocks on the outer, middle, and inner tracks. Run some random requests (e.g., -a -1 -A 5,-1,0, which specifies that random requests should be used via the -a -1 flag and that five requests ranging from 0 to the max be generated), and compute the seek, rotation, and transfer times. Use different random seeds. What is the bandwidth (in sectors per unit time) on the outer, middle, and inner tracks?

8. A scheduling window determines how many requests the disk can examine at once. Generate random workloads (e.g., -A 1000,-1,0, with different seeds) and see how long the SATF scheduler takes when the scheduling window is changed from 1 up to the number of requests. How big of a window is needed to maximize performance? Hint: use the -c flag and don’t turn on graphics (-G) to run these quickly. When the scheduling window is set to 1, does it matter which policy you are using?

9. Create a series of requests to starve a particular request, assuming an SATF policy. Given that sequence, how does it perform if you use a bounded SATF (BSATF) scheduling approach? In this approach, you specify the scheduling window (e.g., -w 4); the scheduler only moves onto the next window of requests when all requests in the current window have been serviced. Does this solve starvation? How does it perform, as compared to SATF? In general, how should a disk make this trade-off between performance and starvation avoidance?

10. All the scheduling policies we have looked at thus far are greedy; they pick the next best option instead of looking for an optimal schedule. Can you find a set of requests in which greedy is not optimal?

# Kapitel 38 – Raid

This section introduces raid.py, a simple RAID simulator you can use to shore up your knowledge of how RAID systems work. See the README for details. Questions

1. Use the simulator to perform some basic RAID mapping tests. Run with different levels (0, 1, 4, 5) and see if you can figure out the mappings of a set of requests. For RAID-5, see if you can figure out the difference between left-symmetric and left-asymmetric layouts. Use some different random seeds to generate different problems than above.

2. Do the same as the first problem, but this time vary the chunk size with -C. How does chunk size change the mappings?

3. Do the same as above, but use the -r flag to reverse the nature of each problem.

4. Now use the reverse flag but increase the size of each request with the -S flag. Try specifying sizes of 8k, 12k, and 16k, while varying the RAID level. What happens to the underlying I/O pattern when the size of the request increases? Make sure to try this with the sequential workload too (-W sequential); for what request sizes are RAID-4 and RAID-5 much more I/O efficient?

5. Use the timing mode of the simulator (-t) to estimate the performance of 100 random reads to the RAID, while varying the RAID levels, using 4 disks.

6. Do the same as above, but increase the number of disks. How does the performance of each RAID level scale as the number of disks increases?

7. Do the same as above, but use all writes (-w 100) instead of reads. How does the performance of each RAID level scale now? Can you do a rough estimate of the time it will take to complete the workload of 100 random writes?

8. Run the timing mode one last time, but this time with a sequential workload (-W sequential). How does the performance vary with RAID level, and when doing reads versus writes? How about when varying the size of each request? What size should you write to a RAID when using RAID-4 or RAID-5?