## Lab Handout 3: Parallel Programming

Students are encouraged to share their ideas in the <u>#lab3</u> Slack channel. SCPD students are welcome to reach out to me directly if they have questions that can't be properly addressed without being physically present for a discussion section.

## Problem 1: Analyzing parallel mergesort

Before starting, go ahead and clone the lab3 folder, which contains a working implementation of mergesort.

```
poohbear@myth15:~$ hg clone /usr/class/cs110/repos/lab3/shared lab3
poohbear@myth15:~$ cd lab3
poohbear@myth15:~$ make
```

Consider the architecturally interesting portion of the mergesort executable, which launches 128 peer processes to cooperatively sort an array of 128 randomly generated numbers. The implementations of createSharedArray and freeSharedArray are omitted for the time being.

```
static void repeatedlyMerge(int numbers[], size_t length, size_t start) {
  int *base = numbers + start;
  for (size_t reach = 2; shouldKeepMerging(start, reach, length); reach *= 2) {
    raise(SIGTSTP);
    inplace_merge(base, base + reach/2, base + reach);
  }
  exit(0);
}

static void createMergers(int numbers[], pid_t workers[], size_t length) {
  for (size_t start = 0; start < length; start++) {
    workers[start] = fork();
    if (workers[start] == 0)
      repeatedlyMerge(numbers, length, start);
  }
}</pre>
```

```
static void orchestrateMergers(int numbers[], pid_t workers[], size_t length) {
 size_t step = 1;
 while (step <= length) {
  for (size_t start = 0; start < length; start += step)
   waitpid(workers[start], NULL, WUNTRACED);
  step *= 2;
  for (size t start = 0; start < length; start += step)
   kill(workers[start], SIGCONT);
}
}
static void mergesort(int numbers[], size_t length) {
 pid_t workers[length];
createMergers(numbers, workers, length);
 orchestrateMergers(numbers, workers, length);
}
static const size t kNumElements = 128;
int main(int argc, char *argv[]) {
 for (size_t trial = 1; trial <= 10000; trial++) {
  int *numbers = createSharedArray(kNumElements);
  mergesort(numbers, kNumElements);
  assert(is_sorted(numbers, numbers + kNumElements));
  freeSharedArray(numbers, kNumElements);
 }
 return 0;
}static bool shouldKeepMerging(size_t start, size_t reach, size_t length) {
  return start % reach == 0 && reach <= length;
}
static void repeatedlyMerge(int numbers[], size_t length, size_t start) {
  int *base = numbers + start;
  for (size t reach = 2; shouldKeepMerging(start, reach, length); reach *= 2) {
```

```
raise(SIGTSTP);
    inplace_merge(base, base + reach/2, base + reach);
  }
  exit(0);
}
static void createMergers(int numbers[], pid_t workers[], size_t length) {
  for (size_t start = 0; start < length; start++) {</pre>
    workers[start] = fork();
    if (workers[start] == 0)
      repeatedlyMerge(numbers, length, start);
  }
}
static void orchestrateMergers(int numbers[], pid_t workers[], size_t length) {
  size_t step = 1;
  while (step <= length) {</pre>
    for (size_t start = 0; start < length; start += step)</pre>
      waitpid(workers[start], NULL, WUNTRACED);
    step *= 2;
    for (size_t start = 0; start < length; start += step)</pre>
      kill(workers[start], SIGCONT);
  }
}
static void mergesort(int numbers[], size_t length) {
  pid_t workers[length];
  createMergers(numbers, workers, length);
  orchestrateMergers(numbers, workers, length);
}
static const size_t kNumElements = 128;
int main(int argc, char *argv[]) {
  for (size_t trial = 1; trial <= 10000; trial++) {</pre>
```

```
int *numbers = createSharedArray(kNumElements);
  mergesort(numbers, kNumElements);
  assert(is_sorted(numbers, numbers + kNumElements));
  freeSharedArray(numbers, kNumElements);
}
return 0;
}
```

The program presented above is a nod to concurrent programming and whether parallelism can reduce the asymptotic running time of an algorithm (in this case, mergesort). We'll lead you through a series of short questions—some easy, some less easy—to test your multiprocessing and signal chops and to understand why the asymptotic running time of an algorithm can sometimes be improved in a parallel programming world.

For reasons I'll discuss shortly, this above program works because the address in the numbers variable is cloned across the 128 fork calls, and this particular address maps to the same set of physical addresses in all 128 processes (and that's different than what usually happens).

The program successfully sorts any array of length 128 by relying on 128 independent processes. In a nutshell, the above program works because:

- All even numbered workers (e.g. workers [0], workers [2], etc.) self-halt, while all odd numbered workers immediately terminate.
- Once all even numbered workers have self-halted, each is instructed to continue on to call inplace\_merge (a C++ built-in) to potentially update the sequence so that numbers[0]<=numbers[1], numbers[2]<=numbers[3], etc. In general, inplace\_merge(first, mid,last) assumes the two ranges [first, mid) and [mid, last) are already sorted in non-decreasing order, and places the merged result in [first,last).
- Once all neighboring pairs have been merged into sorted sub-arrays of length 2, workers[0], workers[4], workers[8], etc. all self-halt while workers[2], workers[6], workers[10], etc. all exit.
- Once all remaining workers self-halt, each is instructed to continue to merge the 64 sorted sub-arrays of length 2 into 32 sorted sub-arrays of length 4.
- The algorithm continues as above, where half of the remaining workers terminate while the other half continue to repeatedly merge larger and larger sub-arrays until only workers[0] remains, at which point workers[0] does one final merge before exiting. The end product is a sorted array of length 128, and that's pretty awesome.

For this lab problem, we want to lead you through a series of short answer questions to

verify a deep understanding of how the entire mergesort system works. Truth be told, the mergesort algorithm as we've implemented is more of theoretical interest than practical. But it's still a novel example of parallel programming that rings much more relevant and real-world than dad-and-pentuplets-go-to-disney.

Use the following short answer questions to guide the discussion. (Note: this entire problem is based on a final exam question from this part autumn quarter.)

- Why is the raise(SIGSTOP) line within the implementation of repeatedlyMerge necessary?
- When the implementation of orchestrateMergers executes the step \*= 2; line the very first time, all worker processes have either terminated or self-halted. Explain why that's guaranteed.
- The repeatedlyMerge function relies on a reach parameter, and the orchestrateMergers function relies on a step parameter. Each of the two parameters doubles with each iteration. What are the two parameters accomplishing?
- Had we replaced the one use of WUNTRACED with a 0, would the overall program still correctly sort an arbitrary array of length 128? Why or why not?
- Had we instead replaced the one use of WUNTRACED with WUNTRACED | WNOHANG instead, would the overall program still correctly sort an arbitrary array of length 128? Why or why not?
- Assume the following implementation of orchestrateMergers replaces the original version. Would the overall program always successfully sort an arbitrary array of length 128? Why or why not?

```
static void orchestrateMergers(int numbers[], pid_t workers[], size_t
length) {
  for (size_t step = 1; step <= length; step *= 2) {
    for (size_t start = 0; start < length; start += step) {
      int status;
      waitpid(workers[start], &status, WUNTRACED);
      if (WIFSTOPPED(status)) kill(workers[start], SIGCONT);
    }
  }
}</pre>
```

• Now assume the following implementation of orchestrateMergers replaces the original version. Note the inner for loop counts down instead of up. Would the overall program always successfully sort an arbitrary array of length 128? Why or why not?

```
static void orchestrateMergers(int numbers[], pid_t workers[], size_t
length) {
```

```
for (size_t step = 1; step <= length; step *= 2) {
   for (ssize_t start = length - step; start >= 0; start -= step) {
     int status;
     waitpid(workers[start], &status, WUNTRACED);
     if (WIFSTOPPED(status)) kill(workers[start], SIGCONT);
   }
}
```

The createSharedArray function (defined in memory.h and memory.cc in your lab3 repo) sets aside space for an array of 128 (or, more generally, length) integers and seeds it with random numbers. It does so using the mmap function you've seen in Assignment 1 and 2, and you'll also saw it a bunch of times while playing with strace last week during discussion section.

The mmap function takes the place of malloc here, because it sets up space not in the heap, but in an undisclosed segment that other processes can see and touch (that's what MAP\_ANONYMOUS and MAP\_SHARED mean).

- Normally virtual address spaces are private and inaccessible to other processes, but that's clearly not the case here. Given what you learned about virtual-to-physical address mapping during this past Monday's lecture, explain what the operating system must do to support this so that only the mergers have shared access but arbitrary, unrelated processes don't?
- Virtual memory is one form of virtualization used so that the above program works. Describe one other form of virtualization you see.
- Assuming the implementation of inplace\_merge is O(n), explain why the running time of our parallel mergesort is O(n) instead of the O(n log n) normally ascribed to the se-

quential version. (Your explanation should be framed in terms of some simple math; it's not enough to just say it's parallel.)

## **Problem 2: Virtual Memory and Memory Mapping**

Assume the OS allocates virtual memory to physical memory in 4096-bytes pages.

- Describe how virtual memory for a process undergoing an execvp transformation might be updated as:
  - the assembly code instructions are loaded
  - the initialized global variables, initialized global constants, and uninitialized global variables are loaded
  - the heap
  - o the portion of the stack frame set up to call main
- If the virtual address 0x7fffa2efc345 maps to the physical page in DRAM whose base address is 0x12345aab8000, what range of virtual addresses around it would map to the same physical page.
- What's the largest size a character array can be before it absolutely must map to three different physical pages?
- What's the smallest size a character array can be and still map to three physical pages?

For fun, optional reading, read these two documents (though you needn't do this reading if you don't want to, since it goes beyond the scope of my lecture-room discussion of virtual memory):

- http://www.cs.cmu.edu/afs/cs/academic/class/15213-f15/www/lectures/ 17-vm-concepts.pdf. These are lecture slides that Bryant, O'Hallaron, and their colleagues rely on while teaching the CMU equivalent of CS110 (they're on a 15-week semester, so they go into more depth than we do).
- <a href="http://www.informit.com/articles/article.aspx?p=29961&seqNum=2">http://www.informit.com/articles/article.aspx?p=29961&seqNum=2</a>: This is an article written some 15 years ago by two senior research scientists at HP Labs who were charged with the task of porting Linux to IA-64.

Also, if it's raining outside and you're short on indoor activity, try the following experiment:

• ssh into any myth machine, and type ps u at the command prompt to learn the process id of your terminal, as with:

```
      poohbear@myth4:~$ ps u

      USER
      PID %CPU %MEM
      VSZ
      RSS TTY
      STAT START
      TIME COMMAND

      poohbear 22628
      0.0
      0.0
      17832
      5796 pts/7
      Ss
      12:35
      0:00 -bash

      poohbear 30578
      0.0
      0.0
      11696
      1236 pts/7
      R+
      16:44
      0:00 ps u
```

The pid of my bash terminal is 22628, but yours will almost certainly be different, of course.
 But assuming a pid 22628, type in the following:

```
poohbear@myth4:~$ cd /proc/22628
poohbear@myth4:/proc/22628$ ls maps
-r--r-- 1 poohbear operator 0 Apr 22 12:39 maps
poohbear@myth4:/proc/22628$ more maps
00400000-004ef000 r-xp 00000000 08:05 1794099
                                                         /bin/bash
006ef000-006f0000 r--p 000ef000 08:05 1794099
                                                         /bin/bash
006f0000-006f9000 rw-p 000f0000 08:05 1794099
                                                         /bin/bash
006f9000-006ff000 rw-p 00000000 00:00 0
0161c000-019d1000 rw-p 00000000 00:00 0
                                                         [heap]
7fe9edc40000-7fe9ee076000 r--p 00000000 08:05 426124
                                                         /usr/lib/locale/
locale-archive
7fe9ee076000-7fe9ee234000 r-xp 00000000 08:05 4440371
                                                         /lib/
x86_64-linux-gnu/libc-2.19.so
7fe9ee234000-7fe9ee433000 ---p 001be000 08:05 4440371
                                                        /lib/
x86_64-linux-gnu/libc-2.19.so
7fe9ee433000-7fe9ee437000 r--p 001bd000 08:05 4440371
                                                         /lib/
x86_64-linux-gnu/libc-2.19.so
7fe9ee437000-7fe9ee439000 rw-p 001c1000 08:05 4440371
                                                         /lib/
x86_64-linux-gnu/libc-2.19.so
7fe9ee439000-7fe9ee43e000 rw-p 00000000 00:00 0
7fe9ee43e000-7fe9ee441000 r-xp 00000000 08:05 4440429
                                                        /lib/
x86_64-linux-gnu/libdl-2.19.so
// many lines omitted for brevity
7fe9eea8d000-7fe9eea8e000 r--p 00022000 08:05 4440389
                                                         /lib/
x86_64-linux-gnu/ld-2.19.so
7fe9eea8e000-7fe9eea8f000 rw-p 00023000 08:05 4440389
                                                         /lib/
x86_64-linux-gnu/ld-2.19.so
7fe9eea8f000-7fe9eea90000 rw-p 00000000 00:00 0
7fffa2efc000-7fffa2f1d000 rw-p 00000000 00:00 0
                                                         [stack]
7fffa2f45000-7fffa2f47000 r-xp 00000000 00:00 0
                                                         [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0 [vsyscall]
poohbear@myth4:/proc/22628$
```

• From the man page for proc: "The proc filesystem is a pseudo-filesystem which provides an interface to kernel data structures. It is commonly mounted at / proc. Most of it is read-only, but some files allow kernel variables to be changed."

Within proc is a subdirectory for every single process running on the machine, and within each of those there are sub-subdirectories that present information about various resources tapped by that process. In my case, the process subdirectory is named 22628, and the sub-subdirectory of interest is maps, which provides information about all of the contiquous regions of virtual memory the process relies on for execution.

To find out what each row and column in the output means, consult this stackoverflow question and read through the accepted answer.

## Problem 3: The Process Scheduler

Recall from Monday's lecture that the process scheduler is a component of the operating system that decides whether a running process should continue running and, if not, what process should run next. This scheduler maintains three different data structures to help manage the selection process: the running queue, the ready queue, and the blocked queue.

- Describe the running queue, ready queue, and the blocked queue. What does each node in each queue consist of?
- Give an example of a system call (with arguments) that may or may not move a running process to the blocked queue.
- Give an example of a system call (with arguments) that is 100% guaranteed to move a process to the blocked queue.
- What needs to happen for a process to be hoisted from the blocked queue to the ready queue?