Lab Solution 3: Parallel Programming

The lab checkoff sheet for all students can be found right here.

Problem 1: Analyzing parallel `mergesort`

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

```
cgregg@myth64:~$ git clone /usr/class/cs110/repos/lab3/shared lab3
cgregg@myth64:~$ cd lab3
cgregg@myth64:~$ 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 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++) {
    workers[start] = fork();
    if (workers[start] == 0)
      repeatedlyMerge(numbers, length, start);
  }
}

// continued on next page
```

```
// continued from previous page

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;
}
```

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 terminate immediately.
- Once all even numbered workers have self-halted, each is instructed to carry 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 deeper understanding of how the entire `mergesort` system works. Truth be told, the `mergesort` algorithm 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.

- Why is the `raise(SIGSTOP)` line within the implementation of `repeatedlyMerge` necessary?
  - *We need all mergers to synchronize with the `main` process, else a merger that's continuing on might advance to merge in a neighboring sub-array before it's been sorted.*
- 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.
  - *When `step *= 2;` is executed for the first time we know that `waitpid` has returned on behalf of every single child process. `waitpid(pid, NULL, WUNTRACED)` only returns because a process either finished or stopped.*
- 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
  - *`reach` tracks the length of the sub-array pairs being merged*
  - *`step` effectively tracks the same thing*
- 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?
  - *No way. `waitpid` would indefinitely block on the 0th worker, since it's self-halting. `WUNTRACED` allows `waitpid` to return on stopped processes, but 0 does 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?

- *No! **`waitpid`** can return a 0 because the child is still executing. If we allow a nonblocking **`waitpid`**, then we might fire a **`SIGCONT`** at a worker than hasn't actually stopped yet, or at one that has stopped even though it's neighboring merger hasn't merged and exited yet.*

- 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);
    }
  }
}
```

- *No. Well intentioned, but the above version allows a merger to carry on and manipulate the sub-array to the right of the one it most recently manipulated, and we don't yet have confirmation this second sub-array has been sorted. We only know it's sorted when its worker exits.*

- 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);
    }
  }
}
```

- *Yes! When a self-halted merger is prompted to continue, we know the worker to its right has already exited, and it can only exit after it's done its final merge. Clever!*

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 also saw it a bunch of times while playing with **strace** last week during discussion section.

```
static int *createSharedArray(size_t length) {
  int *numbers =
    static_cast<int *>(mmap(NULL, length * sizeof(int), PROT_READ | PROT_WRITE,
                            MAP_SHARED | MAP_ANONYMOUS, -1, 0));

  static RandomGenerator rgen;
  for (size_t i = 0; i < length; i++)
    numbers[i] = rgen.getNextInt(kMinValue, kMaxValue);
  return numbers;
}
```

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 Wednesday'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?
  - *The operating system needs to maintain information about the range of virtual addresses introduced by **mmap** and ensure that the same range of virtual addresses in the clones map to the same set of physical pages in main memory, so they're all aliasing the same bytes.*
- Virtual memory is one form of virtualization used so that the above program works. Describe one other form of virtualization you see.
  - *Answer specific to this problem: lots of sorting processes appear as one.*
- 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 sequential version. (Your explanation should be framed in terms of some simple math; it's not enough to just say it's parallel.)
  - Time spent merging is, informally, O(2) + O(4) + … + O(n) -> O(2n) -> O(n). Because *mergers merge in parallel, we incur O(2) cost merging **all** neighboring array pairs of length 1.*
  - *Time spent in orchestrator between merge levels: O(n) + O(n/2) + … + O(1) -> O(n).*

Problem 2: Virtual Memory and Memory Mapping

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

- Describe how virtual memory for a process undergoing an `execvp` transformation might be updated as:
  - the assembly code instructions are loaded
    - *the text segment (called `.text`) is memory mapped to the assembly code instructions packed within the executable file (e.g. `/usr/bin/emacs` or `./diskimageaccess`). The size of the text segment within the executable defines the size of the text segment in the virtual address space. The virtual addresses won't immediately be mapped to physical addresses, but as unmapped virtual addresses are deferenced for their assembly code instructions, the pages around those virtual addresses will be mapped to physical pages which are themselves populated with the assembly code instructions residing there.*
  - the initialized global variables, initialized global constants, and uninitialized global variables are loaded
    - *the data segment (called `.data`) is memory mapped to the data segment of initialized global variables defined with the executable file. Again, the size of the segment is dictated by the side of the data segment within the executable file, and the pages around the virtual addresses of these globals are lazily mapped to physical memory, and the physical memory is populated with the information from the executable.*
    - *the initialized global constants (stored in the `.rodata` segment) are set up the same way, except that the virtual addresses are likely to be marked as read-only.*
    - *the locations of the uninitialized globals are stored in the `.bss` segment of the executable and are automatically assumed to be 0. All of the uninitialized globals are mapped to a zero page that, like the initialized globals, can be updated to be non-zero. If an uninitialized global is updated and the virtual page surrounding it is still mapped to the zero page, then a physical page is allocated, zeroed out, and then used to back the virtual*

page of all zeroes. The write to the uninitialized global is then carried out as if it were a global visibly initialized to 0.

- the heap
  - *Implementation dependent, but the heap would either be of size zero starting out, or a small number of virtual pages would be set and lazily mapped to physical addresses as information is written through heap memory. As the heap is exhausted or fragmented enough that a larger heap is needed, the heap is extended to include even more virtual pages that themselves are lazily mapped to physical pages.*
- the portion of the stack frame set up to call `main`
  - *virtual pages are set aside to store the `argv` and `path` strings passed through and implied by the call to `execvp`, and an initial range of virtual addresses would be established and lazily mapped to physical addresses to store state passed to the `main` function. As `main` calls more and more functions, the portion of the stack in use might grow to the point where even **more** virtual pages need to be mapped to physical memory. Always lazily.*

- If the virtual address `0x7fffa2efc345` maps to the physical page in main memory whose base address is `0x12345aab8000`, what range of virtual addresses around it would map to the same physical page?
  - *Since we're assuming 4096-byte pages, we've expect `0x7fffa2efc000` through `0x7fffa2efcfff` to all map to the physical page with base address `0x12345aab8000` (which houses bytes addressed `0x12345aab8000` through `0x12345aab8fff`).*
- What's the largest size a character array can be before it absolutely must map to three different physical pages?
  - *2 * 4096 + 1 = 8193 bytes. Basic pigeonhole principle.*
- What's the smallest size a character array can be and still map to three physical pages
  - *4096 + 1 + 1 = 4098 bytes. All bytes of the array must be contiguous, but in theory, it's possible (even if unlikely) that the 0th byte of the array is the last byte of one physical page, bytes 1 through 4096 fill a second physical page, and byte 4097 must roll over to the 0th byte of a third physical page.*

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).*
- *http://www.informit.com/articles/article.aspx?p=29961&seqNum=2: 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:*

```
cgregg@myth64:~$ ps u
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
cgregg 22606  0.6  0.0  15716  5004 pts/18   Ss   18:14   0:00 -bash
cgregg 22827  0.0  0.0  30404  1516 pts/18   R+   18:14   0:00 ps u
cgregg@myth64:~$
```

- *The pid of my bash terminal is 22606, but yours will almost certainly be different. But assuming a pid 22606, type in the following*

```
cgregg@myth64:~$ cd /proc/22606
cgregg@myth64:/proc/22606$ ls -lta maps
-r--r--r-- 1 cgregg operator 0 Jan 28 18:15 maps
cgregg@myth64:/proc/22606$ more maps
00400000-004f4000 r-xp 00000000 08:01 2752634                /bin/bash
006f3000-006f4000 r--p 000f3000 08:01 2752634                /bin/bash
006f4000-006fd000 rw-p 000f4000 08:01 2752634                /bin/bash
006fd000-00703000 rw-p 00000000 00:00 0
01230000-01405000 rw-p 00000000 00:00 0                      [heap]
7f7de7c51000-7f7de8069000 r--p 00000000 08:01 8921285        /usr/lib/locale/locale-archive
7f7de8069000-7f7de8229000 r-xp 00000000 08:01 6033975        /lib/x86_64-linux-gnu/libc-2.23.so
7f7de8229000-7f7de8429000 ---p 001c0000 08:01 6033975        /lib/x86_64-linux-gnu/libc-2.23.so
7f7de8429000-7f7de842d000 r--p 001c0000 08:01 6033975        /lib/x86_64-linux-gnu/libc-2.23.so
7f7de842d000-7f7de842f000 rw-p 001c4000 08:01 6033975        /lib/x86_64-linux-gnu/libc-2.23.so
7f7de842f000-7f7de8433000 rw-p 00000000 00:00 0
7f7de8433000-7f7de8436000 r-xp 00000000 08:01 6033979        /lib/x86_64-linux-gnu/libdl-2.23.so
7f7de8436000-7f7de8635000 ---p 00003000 08:01 6033979        /lib/x86_64-linux-gnu/libdl-2.23.so
7f7de8635000-7f7de8636000 r--p 00002000 08:01 6033979        /lib/x86_64-linux-gnu/libdl-2.23.so
7f7de8636000-7f7de8637000 rw-p 00003000 08:01 6033979        /lib/x86_64-linux-gnu/libdl-2.23.so
7f7de8637000-7f7de865c000 r-xp 00000000 08:01 6029424        /lib/x86_64-linux-gnu/libtinfo.so.5.9
7f7de865c000-7f7de885b000 ---p 00025000 08:01 6029424        /lib/x86_64-linux-gnu/libtinfo.so.5.9
7f7de885b000-7f7de885f000 r--p 00024000 08:01 6029424        /lib/x86_64-linux-gnu/libtinfo.so.5.9
7f7de885f000-7f7de8860000 rw-p 00028000 08:01 6029424        /lib/x86_64-linux-gnu/libtinfo.so.5.9
7f7de8860000-7f7de8886000 r-xp 00000000 08:01 6033971        /lib/x86_64-linux-gnu/ld-2.23.so
7f7de8a28000-7f7de8a5d000 r--s 00000000 00:13 713            /run/nscd/dbS6IIxD (deleted)
7f7de8a5d000-7f7de8a61000 rw-p 00000000 00:00 0
7f7de8a7e000-7f7de8a85000 r--s 00000000 08:01 8924731        /usr/lib/x86_64-linux-gnu/gconv/gconv-modules.cache
7f7de8a85000-7f7de8a86000 r--p 00025000 08:01 6033971        /lib/x86_64-linux-gnu/ld-2.23.so
7f7de8a86000-7f7de8a87000 rw-p 00026000 08:01 6033971        /lib/x86_64-linux-gnu/ld-2.23.so
7f7de8a87000-7f7de8a88000 rw-p 00000000 00:00 0
7fff30343000-7fff30364000 rw-p 00000000 00:00 0              [stack]
7fff30382000-7fff30384000 r--p 00000000 00:00 0              [vvar]
7fff30384000-7fff30386000 r-xp 00000000 00:00 0              [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0      [vsyscall]
cgregg@myth64:/proc/22606$
```

- *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* **`22606`**, *and the sub-subdirectory of interest is* **`maps`**, *which provides information about all of the contiguous 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

The Linux Kernel is responsible for *scheduling* processes onto processor cores. 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 running queue keeps track of all of the processes that are currently assigned to the CPU. The nodes in that queue needn't store very much if anything at all, since the CPUs themselves house everything needed for execution. The running queue could be of length 0 (meaning all processes are blocked), or its length can be as high as the number of CPUs.

- The ready queue
  - The ready queue keeps track of all of the processes that aren't currently running but are qualified to run. The nodes in the queue store the state of a CPU at the moment it was pulled off the processor. That information is used to restore a CPU when a process is promoted from ready to running again, so that the process can continue as if it was never interrupted.

- The blocked set
  - This set holds processes which cannot at the moment carry on without some external event happening — they may be waiting for user input, waiting for memory reads, waiting on a network, or waiting for another process to change state. The blocked set looks much like the ready queue, except that it contains processes that were forced off the processor even before its time slice ended. A process is demoted to the blocked set because it blocked on something (e.g. **`waitpid`**).

  - Give an example of a system call (with arguments) that may or may not move a running process to the blocked set.
    - ***`waitpid(-1, NULL, 0)`*** *might block indefinitely if there are child processes but none of them have changed state since the last time* ***`waitpid`*** *was called. It might return immediately if a child finished prior to the* ***`waitpid`*** *call, or if there aren't any child processes at the time of the* ***`waitpid`*** *call.*
  - Give an example of a system call (with arguments) that is 100% guaranteed to move a process to the blocked set.
    - ***`sleep(100)`***. *The process is moved to the blocked set and a timer interrupt is scheduled to lift the process from the blocked set to the ready queue after 100 seconds.*
  - What do you think needs to happen for a process to be hoisted from the blocked set to the ready queue?
    - Some external interrupt must trigger the scheduler to move it up. A timer interrupt for a **`sleep`** call, an I/O interrupt for a **`read`** call, a **`SIGCHLD`** for a **`waitpid`** call, etc.