CS 33:
Introduction to Computer Organization

Week 8

Admin

- Homework 5: “due” Wednesday, November 25th
- Lab 4: OpenMP (AKA your reward for finishing Smashing Lab) “due” Thursday, December 3rd.
- Lab 4: OpenMP (AKA your reward for finishing Smashing Lab) **due** Friday, December 4th.
- Yes, actually, honest to goodness **due** on Friday. That is the DROP DEAD DATE. Any assignments submitted beyond that date will NOT be accepted.

Agenda

- Midterm 2
- Wrap up Processes/Threads and Synchronization
- Lab 4: OpenMP
- HW5/Class Material included in the slides that there's no way in hell we'll get to:
  - Command line args
  - Deadlock Cases
  - Thread safe vs. Reentrancy

Midterm 2

- Mean: 48.1 (out of 99)
- Median: 48 (out of 99)
- Interestingly in my section, something of a bimodal distribution.

Midterm Secret Solutions

- Question 1: Though unconventi
  insist on mea

REDACTED

eyeball of
, the unusual number of gorillas.

Processes

- What happens when a child process completes?
- Processes can be “running”, “stopped”, or “terminated”.
- A completed child process is terminated, but the resources that are used to keep track of the child process still exist.
- This is (and this is completely real), known as a zombie. A zombie child.

Processes

- An undead process is one who has technically terminated.
  - More formal definition of exit(0): terminates the process
- However, you can't get rid of a zombie that easily; the resources that the operating system uses to manage that process continue to exist.
- A process will do so unless, and I'm dead serious, it is “reaped”, after which its resources are reclaimed.

Processes

- One of the functions that the parent has over the child is:
  - waitpid(pid_t pid, int *status, int options)
- From the parent's perspective, the default behavior is “wait for the child process with process id of 'pid' to terminate before continuing”
- If pid == -1, waitpid will wait for any one of the child processes to complete.

Processes

- However, waitpid technically serves an additional role regarding system resources.
- waitpid essentially waits for the child to die and become a zombie so that the parent can reap it of its resources.
- ...which is a terrifying glimpse into the disturbed mind of whoever came up with this.

Processes

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main()
{
  if(fork() == 0)
  {
    printf("a");
  }
  else
  {
    printf("b");
    waitpid(-1, NULL, 0);
  }
  printf("c");
  exit(0);
}
```

- What's the output of this program?

Processes

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main()
{
  if(fork() == 0)
  {
    printf("a");
  }
  else
  {
    printf("b");
    waitpid(-1, NULL, 0);
  }
  printf("c");
  exit(0);
}
```

- Once fork is hit, the child will print a and the parent will print b.
- Afterwards, both will print c.
- The parent will not print c until the child has printed a and c.
- There are no other restrictions on ordering.
- abcc, acbc, bacc

Processes

- Because variables that are created before a fork is duplicated in the child process, this means that child processes also share file descriptors.
- The OS needs to be able to keep track of what files that each process has opened.
- The OS does this by maintaining a table of processes and their corresponding files that are open.
- Each entry has a label or a “file descriptor”

Processes

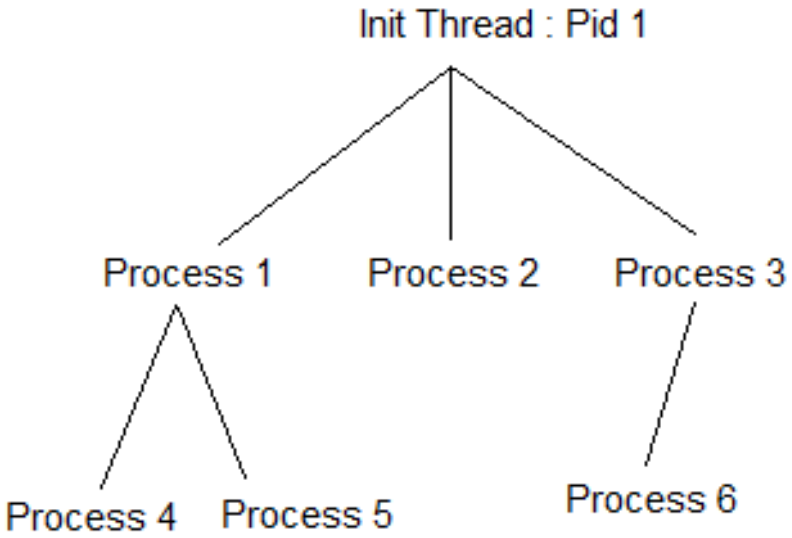
- In order for the process to access the file, it has to go through the OS so it must know the file descriptor of the file it is trying to access.
- A file descriptor is just an integer that corresponds to an index into the OS's table of files.
- Thus, children also have duplicates of the file descriptors and can thus access the same files.

Processes

- File descriptors have to be closed in order to tell the OS that it's done dealing with a particular file.
- As a result, when a child gets a cloned file descriptor, the parent and child are responsible for closing the file descriptor.

Processes

- With Processes, the organization looks like this:
- If a child process becomes orphaned (ex. process 1 exits while process 4 and process 5 are still running), the OS arranges for the process to be adopted by the kindly Mrs. Init Process from down the street.



Processes

- Because of this tree hierarchy and because most processes run in isolation (no shared memory), Processes really only have to worry about the high level status of the children:
  - Is it still running? Paused?
  - How did it exit (normally or via signal)?
  - Why was it out so late last night? Has it been drinking? Was it Braden? You know he's a bad kid.
- As a result, processes don't have to worry much about synchronization. Not so with the hyper-social Threads.

Threads

- All threads that belong to a particular process are peers, which mean there's a lot more management that can occur.
- int pthread_create(pthread_t *tid, pthread_attr_t const *attr, func *f, void *args)
  - Creates thread, assigns thread id to tid. When thread starts, it will call f(args)
- pthread_t pthread_self(void);
  - Get own thread id.
- void pthread_exit(void *thread_return)
  - Thread kills itself with return value in *thread_return

Threads

- int pthread_cancel(pthread_t tid)
  - Thread murders another thread with id tid, in cold blood.
- int pthread_join(pthread_t tid, void ** thread_return)
  - One thread waits for thread tid to complete before continuing.
  - Technically more reaping happens here.

Threads

- int pthread_detach(pthread_t tid)
  - Makes thread specified by tid autonomous. That is, the thread cannot be killed via by pthread_cancel and it cannot be reaped/waited on by pthread_join.
  - The thread achieves invincibility.

Threads

Consider the following code:

```
int main()
{
  while(true)
  {
     pthread_t tid;
     int connfd = Accept(...);
     pthread_create(&tid, 0,
thread, &connfd);
  }
}
```

```
void *thread(void * args)
{
  pthread_detach(pthread_self());
  int * pconnfd = args;
  int connfd = *pconnfd;
  process(connfd);
  close(connfd);
  return 0;
}
```

- Does this work?

Threads

- connfd is a local variable of the main thread.
- Thus, connfd has an address that corresponds to some spot on the main.
- The value of connfd is assigned in the main thread by Accept.
- The address to this value of connfd is passed into the thread.

```
while(true)
{
  pthread_t tid;
  int connfd = Accept(...);
  pthread_create(&tid, 0,
thread, &connfd);
}

void *thread(void * args)
{
  ...
  int * pconnfd = args;
  int connfd = *pconnfd
  ...
}
```

Threads

- The thread then uses the address of the main thread's connfd for args.
- The thread will get the value and store it to a local variable.
- Meanwhile, the main thread re-assigns a value to connfd.
- The address that the old thread uses still corresponds to connfd in the main thread.

```
while(true)
{
  pthread_t tid;
  int connfd = Accept(...);
  pthread_create(&tid, 0,
thread, &connfd);
}

void *thread(void * args)
{
  ...
  int * pconnfd = args;
  int connfd = *pconnfd
  ...
}
```

Threads

- If that reassignment (connfd = Accept...) occurs before connfd = *pconnfd, then the old value will be wiped away.
- This is an example of a race condition (and a pretty bad one).
- Make sure this doesn't happen.

```
while(true)
{
  pthread_t tid;
  int connfd = Accept(...);
  pthread_create(&tid, 0,
thread, &connfd);
}

void *thread(void * args)
{
  ...
  int * pconnfd = args;
  int connfd = *pconnfd
  ...
}
```

Threads

- Conclusion?
- Any time a thread gets a pointer that points to another thread's stack frame, be wary.
- We can fix this by making it so that the value passed into the thread is NOT a pointer pointing to the main thread's stack frame.

```
while(true)
{
  pthread_t tid;
  int connfd = Accept(...);
  pthread_create(&tid, 0,
thread, &connfd);
}

void *thread(void * args)
{
  ...
  int * pconnfd = args;
  int connfd = *pconnfd
  ...
}
```

Threads

- Now, we deal with an int *, whose address IS still in the main thread's stack.
- However, what we pass into the thread is the value of the int*, which is malloc'd.

```
while(true)
{
  pthread_t tid;
  int *connfd = malloc(sizeof(int));
  *connfd = Accept(...);
  pthread_create(&tid, 0, thread,
connfd);
}
void *thread(void * args)
{
  ...
  int * pconnfd = args;
  int connfd = *pconnfd
  ...
}
```

Synchronization + Race Conditions

- One of the major advantages of using threads over processes is the ease by which threads can share data.
- This of course, is marred by the potential that by using threads, you get the wrong result.
- That's bad.
- How do we ensure that behavior is correct, even when the execution order is not guaranteed?

Synchronization + Race Conditions

- We can enforce protection for shared variables or critical sections with semaphores, which are basically locks that tell us how many threads can enter a section of code.
- The semaphore is of type: sem_t and it is sort of a glorified counter or more conceptually, a door that allows/prevents entry into a section of code.

Synchronization + Race Conditions

- While the counter is non-zero, the door is open and thread can pass. When this happens, the counter decrements.
- When the counter is zero, the door is closed. The thread must wait for the door to be open again (counter becomes non-zero) before it can pass.

Synchronization + Race Conditions

- sem_t sem;
- sem_wait(&sem) – If sem is non-zero, return and decrement sem (the door is open, thread is allowed to pass). If sem is zero, wait until sem is non-zero, then decrement and return (the door is closed, wait for it to open).
- sem_post(&sem) – Increment sem by one. If sem was previously zero, the door was closed. Open the door and allow another thread in.

Synchronization + Race Conditions

- Say we have the following line of code:
  - <line of code>
- ...and we want to allow only two threads to be able to execute that line of code at any time.
- sem_t sem;
- sem_init(&sem, 0, 2); //0 is an options argument, 2 is the number of threads allowed
- sem_wait(&sem);
- <line of code>;
- sem_post(&sem);

Synchronization + Race Conditions

- Let's say we have some shared resource that you can read and write from. This can be done via two functions:

```
void read()
{
  <read shared resource>
}

void write()
{
  <write shared resource>
}
```

Synchronization + Race Conditions

- However, we want the following conditions:
  - There can be an unbounded number of concurrent readers.
  - There can be only 1 writer and if someone is writing, there can be no readers.
- How can we go about this?

Synchronization + Race Conditions

```
void read()
{
 <read>
}
```

```
void write()
{
 <write>
}
```

Synchronization + Race Conditions

```
sem_t r; // init to ?
sem_t w;// init to 1
void read()
{
  sem_wait(&r);
  <read>
  sem_post(&r);
}
```

```
void write()
{
  sem_wait(&w);
  <write>
  sem_post(&w);
}
```

- We'll definitely want some semaphores around the critical sections in read/write.
- Is this right?

Synchronization + Race Conditions

```
sem_t r; // init to ?
sem_t w;// init to 1
void read()
{
  sem_wait(&r);
  <read>
  sem_post(&r);
}
```

```
void write()
{
  sem_wait(&w);
  <write>
  sem_post(&w);
}
```

- This prevents multiple writers, but it allows writing to happen at the same time as reading.
- Also, what do we initialize r to We want unbounded readers.
- Let's try to solve the “writing/reading at the same time” problem first

Synchronization + Race Conditions

```
sem_t r; // init to ?
sem_t w;// init to 1
void read()
{
  sem_wait(&r)
  sem_wait(&w)
  <read>
  sem_wait(&w)
  sem_wait(&r)
}
```

```
void write()
{
  sem_wait(&w);
  <write>
  sem_post(&w);
}
```

- Now, writing cannot happen at the same time as reading.
- What's the next step?

Synchronization + Race Conditions

```
sem_t r; // init to ?
sem_t w;// init to 1
void read()
{
  sem_wait(&r)
  sem_wait(&w)
  <read>
  sem_wait(&w)
  sem_wait(&r)
}
```

```
void write()
{
  sem_wait(&w);
  <write>
  sem_post(&w);
}
```

- If we initialized r to, say 10, we could not have 10 readers reading at the same time because w is initialized to 1.
- We recognize that we only want to run sem_wait(&w) when there are no readers.

Synchronization + Race Conditions

```
sem_t r; // init to ?
sem_t w;// init to 1
void read()
{
  sem_wait(&r)
  sem_wait(&w)
  <read>
  sem_wait(&w)
  sem_wait(&r)
}
```

```
void write()
{
  sem_wait(&w);
  <write>
  sem_post(&w);
}
```

- If no readers, there might be a writer. Therefore, a reader must call sem_wait(&w) to reserve the critical section.
- If there are readers, then that means there cannot be a writer.
- Therefore, the reader doesn't need to call sem_wait(&w)

Synchronization + Race Conditions

```
sem_t r; // init to ?
sem_t w;// init to 1
void read()
{
  sem_wait(&r)
  if(there are no readers)
    sem_wait(&w)

  <read>

  if(this is the last reader)
    sem_post(&w)
  sem_post(&r)
}
```

```
void write()
{
  sem_wait(&w);
  <write>
  sem_post(&w);
}
```

- Once a reader has finished reading, it must also check: if it is the last reader, then it should be able to allow a writer. Therefore it must release the lock by calling sem_post(&w)

Synchronization + Race Conditions

```
sem_t r; // init to ?
sem_t w;// init to 1
int reader_count = 0;
void read()
{
  sem_wait(&r)
  reader_count++;
  if(read_count == 1)
    sem_wait(&w)

  <read>

  reader_count--;
  if(reader_count == 0)
    sem_post(&w)
  sem_post(&r)
}
```

```
void write()
{
  sem_wait(&w);
  <write>
  sem_post(&w);
}
```

- Does this work?

Synchronization + Race Conditions

```
sem_t r; // init to ?
sem_t w;// init to 1
int reader_count = 0;
void read()
{
  sem_wait(&r)
  reader_count++;
  if(read_count == 1)
    sem_wait(&w)

  <read>


  reader_count--;
  if(reader_count == 0)
    sem_post(&w)
  sem_post(&r)
}
```

```
void write()
{
  sem_wait(&w);
  <write>
  sem_post(&w);
}
```

- Two problems, first of all, this only works if we can set r to infinity, which we can't.
- Second, there's still a possible race condition. What if two reader threads increment reader_count before either can get to reader_count == 1? The whole system messes up.

Synchronization + Race Conditions

```
sem_t r; // init to ?
sem_t w;// init to 1
int reader_count = 0;
void read()
{
  sem_wait(&r)
  reader_count++;
  if(read_count == 1)
    sem_wait(&w)

  <read>

  reader_count--;
  if(reader_count == 0)
    sem_post(&w)
  sem_post(&r)
}
```

```
void write()
{
  sem_wait(&w);
  <write>
  sem_post(&w);
}
```

- That this treats the <read> as the critical section, but that's not what we want to protect.
- We want to make sure that infinite readers can read, but only one reader can increment reader_count at a time.

Synchronization + Race Conditions

```
sem_t r; // init to 1
sem_t w;// init to 1
int reader_count = 0;
void read()
{
  sem_wait(&r);
  reader_count++;
  if(read_count == 1)
    sem_wait(&w);
  sem_post(&r);
  <read>
  sem_wait(&r);
  reader_count--;
  if(reader_count == 0)
    sem_post(&w)
  sem_post(&r)
}
```

```
void write()
{
  sem_wait(&w);
  <write>
  sem_post(&w);
}
```

- It satisfies the constraints (single writer, multiple reader).
- Is there maybe still a problem though?

Synchronization + Race Conditions

```
sem_t r; // init to 1
sem_t w;// init to 1
int reader_count = 0;
void read()
{
  sem_wait(&r);
  reader_count++;
  if(read_count == 1)
    sem_wait(&w);
  sem_post(&r);
  <read>
  sem_wait(&r);
  reader_count--;
  if(reader_count == 0)
    sem_post(&w)
  sem_post(&r)
}
```

```
void write()
{
  sem_wait(&w);
  <write>
  sem_post(&w);
}
```

- This method is unfair to writers.
- That is to say, a potential writer could be starved since if readers keep coming in, the writer will never have a chance to write.

Lab 4: OpenMP Lab

- Unlike the other labs, the objective of this lab is simple.
- Premise: You have code that's slow.
- Goal: Make it go fast.
- The End.

Lab 4: OpenMP Lab

- Getting started:
  - Download the openmplab.tgz file from CCLE
  - Copy it to lnxsrv09.seas.ucla.edu (more than ever, USE lnxsrv09 and gcc version 5.2.0)
  - Unzip it.
- To run the unaltered code, use:
  - make seq
  - This will produce an executable called “seq” (note, the specs are wrong).

Lab 4: OpenMP Lab

- In order to speed it up, you are to use two tools, one familiar and one new (probably):
  - To determine what to speed up: gprof.
  - To speed it up: OpenMP, an API for shared memory, thread based parallelism.

Lab 4: OpenMP Lab

- To use gprof:
- Compile with GPROF=1
  - ex. make seq GPROF=1
- Execute the executable (this produces gmon.out)
  - ex. ./seq
- Run gprof with the executable as the argument
  - ex. gprof seq | less
  - The command gprof seq normally prints out a wall of text. The “| less” redirects the output as input into the “less” command, which allows you to read it more easily.
  - Quit “less” with q.

Lab 4: OpenMP Lab

- The code provided performs a computation using six functions (func0 to func5) located in the func.c.
- You are to speed up the execution of the program by speeding up these six function.
- How?

Lab 4: OpenMP Lab

- You could use the normal Chapter 5 style optimizations, but we want something powerful that makes uses of multiple processors.
- This means using threads and doing multithreading.
- However, we want to use something a little more powerful and exotic than POSIX threads.
- So exotic that you might never see it again...

Lab 4: OpenMP Lab

- As a result, we use OpenMP
- OpenMP is an extension to C/C++ (and Fortran) that allows for a simple and convenient way of launching threads to do work in parallel among multiple processors.
- Surprisingly, you won't be personally using a bunch of new functions.
- You'll be specifying a bunch of pre-processor directives.

Lab 4: OpenMP Lab

- Yup, these: “#”.
- As a refresher, preprocessor directives are statements that are marked with “#”.
- Statements marked with these are instructions that tell the processor what to do:
  - #define min(a, b) (a < b ? a : b)
  - #include <stdio.h>
  - #don't become sentient

Lab 4: OpenMP Lab

- The first step of compilation is the preprocessing step where the compiler will find all of the directives and follow the instructions.
- OpenMP exposes an interface for multithreading that is almost purely based on these preprocessor instructions.
- Other than some helper functions defined in omp.h. Ex.
  - omp_set_num_threads(int);
  - omp_get_thread_num();

Lab 4: OpenMP Lab

- Ex:

```
int main()
{
  int a = 0;
  printf(“%d\n”, a);
}
```

- Say you want to parallelize this so that 4 threads run this code.

Lab 4: OpenMP Lab

Ex:

```
int main()
{
  omp_set_num_threads(4); //Use 4 threads
  #pragma omp parallel //Define the following
  {                                  //block as parallel code
    int a = 0; //run by each thread
    printf("%d\n", a);
  }
}
```

Lab 4: OpenMP Lab

Ex:

```
int main()
{
  #pragma omp parallel
  {
    int a = 0;
    printf("%d\n", a);
  }
}
```

- If the number of threads is not specified, the default is used. On SEASnet lnxsrv09, this means 32 (the number of processors).

Lab 4: OpenMP Lab

- Upon hitting the block specified by the #pragma omp directive, the single thread of execution branches off into several parallel threads.
- When each thread reaches the end of the block:

  ```
  #pragma omp parallel
  {
    ...
  } ← Here
  ```

- ...each thread will wait for all of the other threads before continuing. This is known as a “barrier”.

Lab 4: OpenMP Lab

- The “parallel” keyword means “launch threads in parallel”
- This the basis how OpenMP works.
- Synchronization and control can also be done via these directives. Within a parallel block, you can also have other directives that specify things.
- Such as...

Lab 4: OpenMP Lab

- critical:

```
#pragma omp critical
{
  <CRITICAL SECTION>
}
```

  - Only one thread can enter the critical section at a time. Other threads must wait (like semaphores)
- atomic:

```
#pragma omp atomic
{
  tmp += 1;
}
```

  - Only one thread can perform a particular read/write. This only applies to single variable that is read/write.

Lab 4: OpenMP Lab

- barrier:
  - #pragma omp barrier
  - All threads must wait here for all threads to complete before continuing.
- ...and so much more.
- So... so much more.
- http://openmp.org/mp-documents/omp-hands-on-SC08.pdf

Lab 4: OpenMP Lab

- Back to this example:

```
int main()
{
  #pragma omp parallel
  {
    int a = 0;
    printf("%d\n", a);
  }
}
```

- What if a were declared outside of the parallel section?

Lab 4: OpenMP Lab

- Back to this example:

```
int main()
{
  int a = 0;
  #pragma omp parallel
  {
    a++;
    printf(“%d\n”, a);
  }
}
```

- Is 'a' shared among the 32 threads?

Lab 4: OpenMP Lab

- Data sharing:
  - By default, a variable declared inside of a parallel block is private to each block.
  - By default, a variable declared outside of a parallel block is shared among all of the threads of a parallel block that accesses the variable. In the previous example, “a” was the same “a” among all threads.
  - “a” was also the same “a” that was declared before entering the parallel block
  - What do we expect to see as an output?

Lab 4: OpenMP Lab

- Data sharing:
  - Utter chaos
  - Because there is no protection of the shared variable and because writes are not atomic, the results are going to be crazy.
  - If we want to share the variables in a way that's more graceful and predictable, we'll have to use critical or atomic but then, we'll essentially be doing code in serial.

Lab 4: OpenMP Lab

- A variable within a parallel block can be made private to each thread by using the “private” keyword. Note: this example is deliberately problematic. To see why, turn the page...

```
int main()
{
  int a = 0;
  #pragma omp parallel private(a)
  {
    a++;
    printf(“%d\n”, a);
  }
}
```

Lab 4: OpenMP Lab

- Warning: the variable will also be private relative to the original declaration.

```
int main()
{
  int a = 123;
  #pragma omp parallel private(a)
  {
    a++;
    printf("%d\n", a);
  }
}
```

- The "a" inside the block will not be initialized to 123. It will be uninitialized.

Lab 4: OpenMP Lab

- If you do want to make the private variable in the block initialized to the value specified in the original thread, use “firstprivate”.

```
int main()
{
  int a = 123;
  #pragma omp parallel firstprivate(a)
  {
    a++;
    printf(“%d\n”, a);
  }
}
```

- The “a” inside the block will now be initialized to 123.

Lab 4: OpenMP Lab

- Loops
- Consider:

```
int i;
for(i = 0; i < N; i++)
{
  <PARALLELIZABLE CODE>
}
```

- You can do this with an “#pragma omp parallel” block, but you'll have to manually assign threads to iterations of the loop.

Lab 4: OpenMP Lab

- Loops
- Or you can do this:

```
int i;
#pragma omp parallel for
for(i = 0; i < N; i++)
{
  <PARALLELIZABLE CODE>
}
```

- ...which magics it up for you.
- However, there are (at least) two bits of trickiness here.

Lab 4: OpenMP Lab

- What if you have this:

```
int acc = 0;
int i;
#pragma omp parallel for
for(i = 0; i < N; i++)
{
  acc += <PARALLELIZABLE CODE>
}
printf("%d\n", acc);
```

- What's the problem with this?

Lab 4: OpenMP Lab

- What if you have this:

```
int acc = 0;
int i;
#pragma omp parallel for
for(i = 0; i < N; i++)
{
  acc += <PARALLELIZABLE CODE>
}
printf("%d\n", acc);
```

- What's the problem with this?
  - acc += … is not atomic. The result will not be correct.

Lab 4: OpenMP Lab

- To ensure that you can use accumulators use the “reduction” keyword, which allows you to collect results into one shared variable among threads.

```
int acc = 0;
int i;
#pragma omp parallel for reduction(+:acc)
for(i = 0; i < N; i++)
{
  acc += <PARALLELIZABLE CODE>
}
printf(“%d\n”, acc);
```

- Format: (<operation> : <variable>);

Lab 4: OpenMP Lab

- Finally, consider:

```
int acc = 0;
int j = 0;
int i;
#pragma omp parallel for reduction(+:acc)
for(i = 0; i < N; i++)
{
 j += 2;
 acc += <PARALLELIZABLE CODE>
}
printf("%d\n", acc);
```

- It looks like j is in a similar situation as acc.

Lab 4: OpenMP Lab

- However, the dependence on j is artificially loop carried. You can covert it to:

```
int acc = 0;
int j = 0;
int i;
#pragma omp parallel for private(j) reduction(+:acc)
for(i = 0; i < N; i++)
{
  j = 2*i;
  acc += <PARALLELIZABLE CODE>
}
printf("%d\n", acc);
```

- You may find that small tweaks like this may be necessary to make an "omp parallel for" possible.

Lab 4: OpenMP Lab

- Final Notes:
- The specs specify methods of invoking the make file, several of which are wrong.
- “make seq” will generate an executable called “seq”.
- “make omp” will generate an executable called “omp”.
- “make ___ SRC=funcv2.c” does not work.
  - If you'd like this functionality to work, in the Makefile change:
    - SRC = filter.c
    - SRCS = $(SRC) main.c func.c util.c
  - to
    - SRC = func.c
    - SRCS = $(SRC) main.c filter.c util.c

Lab 4: OpenMP Lab

- Final Notes:
  - The run times will vary each time you call filter.
  - As it turns out, as the SEASnet processor become more burdened, it will actually slow itself down, which will lead to a slower execution time.
  - If you think you have a solution, double check your time by running your code repeatedly.
  - You are supposed to speed up the “FUNC TIME” as reported by the print statement by 3.5x.

Lab 4: OpenMP Lab

- Final Notes:
  - When you run the code with GPROF=1, the times will slow down. Example output:

    FUNC TIME: .670985

    TOTAL TIME: 2.772028
  - This suggests that the total amount of time spent by the functions in func.c take up ~20-25%.
  - If you actually look at the gprof print out, you may see this:

Lab 4: OpenMP Lab

- Final Notes:

```
%    cumulative      self     self    total
time    seconds  seconds    calls ms/call ms/call  name
61.19      0.44     0.44       15   29.37   31.65  func1
25.03      0.62     0.18  5177344    0.00    0.00  rand2
 2.78      0.64     0.02   491520    0.00    0.00  f...
 2.78      0.66     0.02        2   10.01   10.01  init
  ...
```

- This suggests that func1 alone is taking 61% of the time.
- I'm no expert, but I think 20 is like, not the same as 61.

Lab 4: OpenMP Lab

- As it turns out, gprof doesn't take into account blocking I/O time, which is the bulk of what the non-func.c functions do.
- Use either your own measurements (get_time) or use the ones that are printed out by the output.

Lab 4: OpenMP Lab

- Final Notes: Memory Leaks
- According to checkmem, OpenMP loves to leak memory.

```
int a;
#pragma omp parallel
{
  a = 0;
}
```

- Even this “leaks memory”.

Lab 4: OpenMP Lab

- Final Notes:
- If the memory error printed out by “make checkmem” is “Memory not freed”, don't worry about it. Otherwise, go back and double check.
- Make sure to use “make check” to confirm that the output.txt generated by the program is the same as correct.txt. If silent, the files are the same.