CS246
C: parallel

May 3

Lab

- Write a program that can take 8, 3 bit numbers and output a string of 24 bits
  - For example:
    - a.out 0 1 2 3 4 5 6 7
    - 000001010011100101110111
  - (read each number as a char, then take only the 3 least significant bits from the char)
- You can just accumulate the bits in a character string or write directly to stdout.

- For an additional complication (not required or even recommended)
  - write the 8 3 bit numbe
  - rs into 3 8 bit numbers and then output those numbers. This will mean that some 3 bit number are represented across 2 8 bit numbers.

My Solution

```
int main(int argc, char const *argv[])
{
    unsigned char output = 0;
    unsigned int res = 0;
    int b = 0;
    for (int i = 1; i <= 8; i++)
    {
        int num = argv[i][0] - ‘0’; // atoi(argv[i]);
        for (int j = 0; j < 3; j++)
        {
            int v = (num & 4) ? 1 : 0;
            output <<= 1;
            printf("%d", v);
            output |= v;
            num <<= 1;
            b++;
            if (b>=8) {
                printf("\n%d\n", output);
                b = 0;
                output = 0;
            }}}
    printf("\n");
    return 0;
```

My Other Solution

- Rather than shifting bits in the number being scanned here I shift bits in the mask
- rather than shifting bits in the answer, shift bits in the thing that might be put in the answer
- maybe more readable

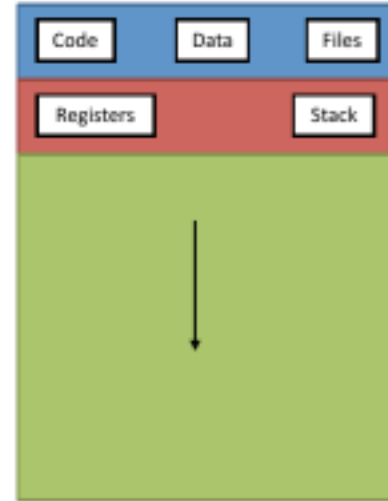
```
int main(int argc, char const *argv[])
{
    unsigned char output = 0;
    unsigned int res = 0;
    int placer = 1 << 7;
    for (int i = 1; i <= 8; i++)
    {
        int num = atoi(argv[i]);
        int mask = 1 << 2; // or just mask = 4;
        for (int j = 0; j < 3; j++)
        {
            int v = (num & mask);
            printf("%d", v);
            if (v)
                output |= placer;
            mask >>= 1;
            placer >>= 1;
            if (!placer) {
                printf("\n%d\n", output);
                placer = 1 << 7;
                output = 0;
            }
        }
    }
    printf("\n");
    return 0;
}
```

Parallel Programming (in C)

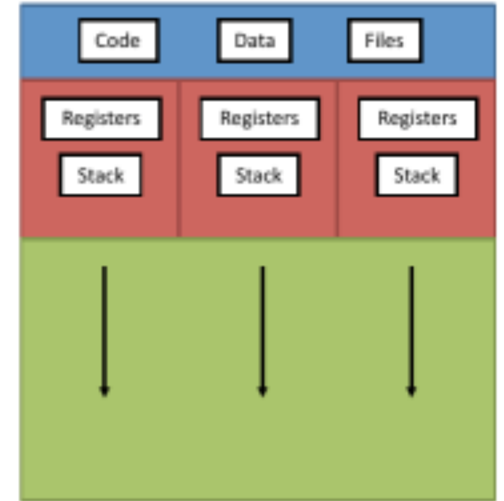
- 2 main ways to do parallel
    - by process
        - multiple independent jobs
    - by thread
        - parallelism within a single job

- Choice is dependent on task and user style

- Why do parallel?
    - I have 32 cores on my machine ….
    - I have 300 servers down the the server room …



Single-threaded



Multi-threaded

Process vs Thread

Process	Thread
Processes are heavy-weight operations.	Threads are lighter-weight operations.
Processes can start new processes using e.g. fork() (system call).	A process can start several threads using e.g pthread_create() (system call).
Each process lives in its own memory (address) space and holds a full copy of the program in memory which consume more memory. Processes don't share memory with other processes.	Threads share memory with other threads of the same process. Threads within the same process live within the same address space, and can thus easily access each other's data structures. The shared memory heaps and pools allow for reduced overhead of shared components.
Inter-process communication is slow as processes have different memory addresses.	Inter-thread communication can be faster than inter-process communication because threads of the same process share memory with the process they belong to.
Context switching between processes is more expensive.	Context switching between threads of the same process is less expensive.

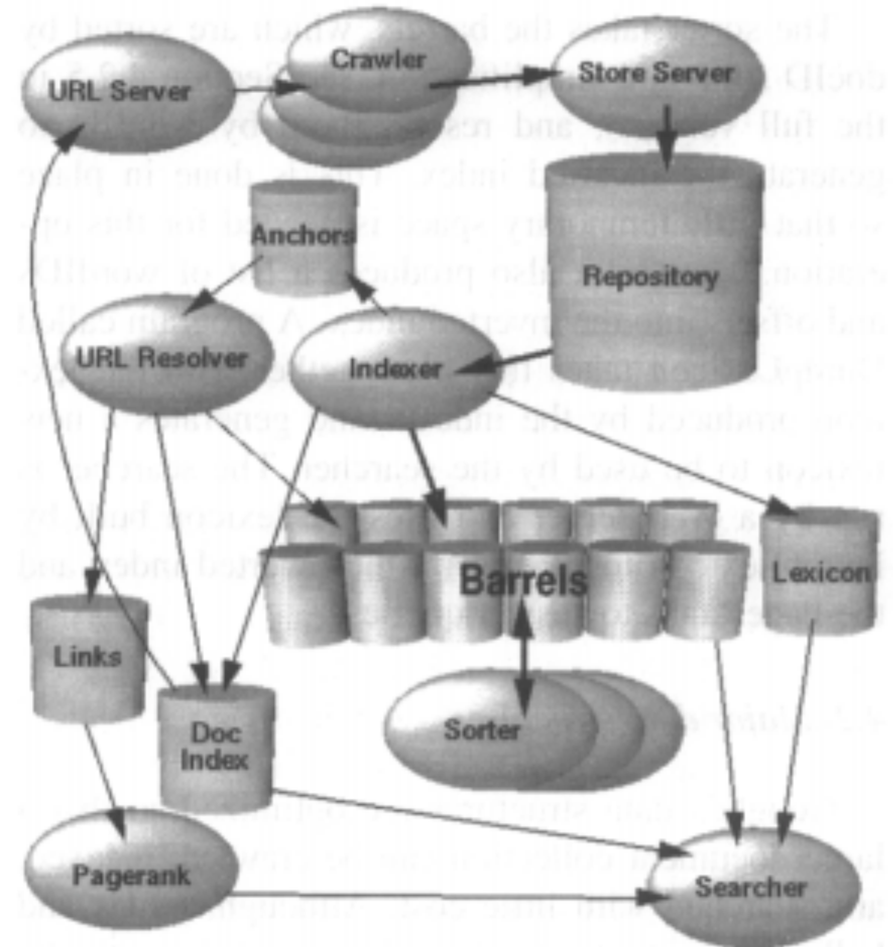
Also, processes can be on different machines in different places

Why not parallel

- Overhead
  - synchronization
  - accessing of shared resources
- Hazards
  - deadlocks / threadlocks
    - A is waiting for B, and B is waiting for A
  - Race Conditions
    - if A before B get C, if B before A get D
- Debugging

Process Parallelism

- Allows for huge scale
  - consider Google and the web crawling problem
  - Each oval represents a process type
    - each type has 1 or more actual processes
  - The processes communicate by reading and writing files



Problem: Perfect numbers

- a “perfect number” is one for which the factors, excluding itself sum to the number
- 6 = 1+2+3
- 28 = 1+2+4+7+14

- Euclid proved a formation rule (IX.36) whereby N=q(q+1)/2 is an even perfect number whenever q is a prime of the form 2^P-1 for prime P — a Mersenne prime. Two millennia later, Euler proved that all even perfect numbers are of this form.

- Nice but we are going to do brute force

Brute Force perfect numbers

```
int main(int argc, char const *argv[])
{
    perfect.count = 0;

    curr = atol(argv[1]);
    long max = atol(argv[2]);
    while (1) {
        checkit();
        if (curr > max)
            break;
    }
    pres("perfect", perfect);
    printf("Complete %s %s\n", argv[1], argv[2]);
    return 0;
}
```

```
typedef struct {
    int count;
    int data[1000];
} accumul;
void pres(char *head, accumul dat)
{
    for (int i = 0; i < dat.count; i++)
        printf("%s %2d %d\n", head, i, dat.data[i]);
}

accumul perfect;
long curr = 0;

void checkit() {
    int val = curr++;
    int j = sqrt(val);
    int sm = 1;
    for (int k = 2; k <= j; k++) {
        int div = val / k;
        if (k*div == val) {
            sm += k;
            if (k!=div)
                sm += div;
        }
    }
    if (sm == val) {
        perfect.data[perfect.count++] = val;
    }
}
```

do not put in 5 twice for 25

Process Parallelism

- If on single machine then useless to start more processes than the machine can support
    - how to know?
        - cat /proc/cpuinfo (| grep processor) (tail -1)
    - or just UNIX> nproc

- Then what???
    - A: a shell script!!!!

Shell script for process parallelism

- by default starts as many processes as there are CPUs on machine
- While loop waits for processes to finish before shell script competes
- can even start on other machines using ssh
    - UNIX> ssh powerpuff “ls”
        - would run ls command on powerpuff (which is a name set up in my .ssh/config file)
- See process.sh in VSC

Thread-level parallelism

- added to C in C11
    - by which time there were a lot of independently developed approaches, many of which got included in C11
        - LOTS of ways in standard C, I will discuss one
    - pthread
        - POSIX threads
            - Portable Operating System Interface
    - atoms
    - locks

Thread saftey

- When dealing with threaded programs that share data structures need changes made by one thread can affect the run of another thread

Global	Thread 1	Thread 2
int *a;		
	a=malloc(1*sizeof(int));	
		printf(“%d\n”, *a);
		free(a);
	printf(“%d\n”, *a);	

Free can never be made truly “thread safe”

Thread saftey is time consuming

- Java often has two classes that are identical except that one “thread-safe” and one is not
  - Vector and ArrayList
- ArrayList NOT safe but much faster
  - Synchronized
    - only one operation allowed to happen at a time

C creating multiple threads

- gcc -pthread
    - need this arg when compiling

- pthread_create
    - takes 4 args
    - pointer to a struct to hold info about the thread
    - NULL == create with default “attributes”
    - the function to start
    - Argument to the function
- pthread_join
    - blocks as long as the thread is running

- Note that two loops here a very similar to the shell script loops

```
pthread_t threads[NTHREADS];
for (int i = 0; i < NTHREADS; i++)
{
    tInfo[i] = (threadInfo){.threadNum = i};
    pthread_create(threads + i, NULL, threadFunc, NULL)
}
for (int i = 0; i < NTHREADS; i++) {
    pthread_join(threads[i], NULL);
}
```

pointer to pthread_t struct

Start threads

pthread_t struct

Why not a pointer here?

Wait for threads to complete

Simple Thread Example

```c
void * myThreadFun(void *vargp) {
    int *ti = vargp;
    printf("started thread %d\n", *ti);
    sleep(2);
    printf("finishing thread %d\n", *ti);
}
int main(int argc, char const *argv[]) {
    int i;
    int N = atoi(argv[1]);
    // Let us create N threads
    pthread_t threads[N];
    for (i = 0; i < N; i++)
    {
        pthread_create(threads+i, NULL, myThreadFun, &i);
    }
    for (int i = 0; i < N; i++) {
        pthread_join(threads[i], NULL);
        printf("Joined %d\n", i);
    }
    pthread_exit(NULL);
    return 0;
}
```

```
UNIX> gcc -pthread pp.c
UNIX> a.out 5
started thread 1
started thread 2
started thread 3
started thread 4
started thread 5
finishing thread 5
finishing thread 5
finishing thread 5
finishing thread 5
finishing thread 5
Joined 0
Joined 1
Joined 2
Joined 3
Joined 4
UNIX>
```

threadFunc for perfect number

- Does the work of the thread
- takes a single argument
  - declared as a void*
  - So almost always a struct that was filled just before thread started
- So could just take checkit code, add a void* param and go!

```
void checkit(void * argum) {
    int val = curr++;
    int j = sqrt(val);
    int sm = 1;
    for (int k = 2; k <= j; k++) {
        int div = val / k;
        if (k*div == val) {
            sm += k;
            if (k!=div)
                sm += div;
            }
        }
    if (sm == val) {
        perfect.data[perfect.count++] = val;
    }
}
```

Thread problems

Multiple threads could try to change this add the same time

Multiple thread could try to change this add the same time

Multiple thread could try to change this add the same time

```
void checkit(void * argum) {
    int val = curr++;
    int j = sqrt(val);
    int sm = 1;
    for (int k = 2; k <= j; k++) {
        int div = val / k;
        if (k*div == val) {
            sm += k;
            if (k!=div)
                sm += div;
        }
    }
    if (sm == val) {
        perfect.data[perfect.count++] = val;
    }
}
```

mutex locks

- Idea: say that a part of the code can only run on one thread at a time.
  - Any other thread that wants to use must wait for the current thread to complete
- Separate lock for each thing being protected
  - I like putting protected sections in own funcs

```
pthread_mutex_t mutexAV;
pthread_mutex_t mutexGN;

void addValue(accumul *addTo, int val) {
    pthread_mutex_lock(&mutexAV);
    addTo->data[addTo->count++] = val;
    pthread_mutex_unlock(&mutexAV);
}

int getNumAndIncr() {
    pthread_mutex_lock(&mutexGN);
    int val = curr;
    curr++;
    pthread_mutex_unlock(&mutexGN);
    return val;
}

// IN MAIN
    pthread_mutex_init(&mutexAV, NULL);
    pthread_mutex_init(&mutexGN, NULL);
```

Atomic Variables

- getNumAndIncr (and the like) is really common
- So rather than rewriting all the time
  - Atomic variables
- #include <stdatomic.h>

- _Atomic long g = ATOMIC_VAR_INIT(0);

- functions like atomic_fetch_add
  - even better — gives you thread-safe ++, - -,
  - Atomics are faster than rolling you own with locks

Threaded Perf Finder

- Base code
  - no particular need for accumul struct
  - threadInfo struct will be passed into each thread
    - just allows thread to know its number
- update to perfect number store is lock protected
- number being checked is atomic

```
typedef struct {
    char header[20];
    int count;
    int data[1000];
} accumul;
typedef struct {
    int threadNum;
} threadInfo;
accumul perfect;
int N = 5;
_Atomic long curr = ATOMIC_VAR_INIT(0);
pthread_mutex_t mutexAV;
void addValue(accumul *addTo, int val) {
    pthread_mutex_lock(&mutexAV);
    addTo->data[addTo->count++] = val;
    pthread_mutex_unlock(&mutexAV);
}
void pres(accumul* dat) {
    printf("%s  %d\n", dat->header, dat->count);
    for (int i = 0; i < dat->count; i++)
        printf("     %2d %d\n", i, dat->data[i]);
}
```

Thread worker

- threads keep going until reaching the target number
- communication between threads is just through curr variable

```
void* checkit(void * args) {
    threadInfo *tInfo = args;
    while (1) {
        long val = curr++; // thread safe!!
        if (val % 1000000 == 0)
            printf("Thread:%2d  number:%10ld\n", tInf
        if (val > N) break;
        int j = sqrt(val);
        int sm = 1;
        for (int k = 2; k <= j; k++) {
            int div = val / k;
            if (k * div == val) {
                sm += k;
                if (k != div)
                    sm += div;
            }
        }
    //printf("%d   %d\n", val, sm);
    if (sm == val) {
        addValue(&perfect, val);
    }}
    return NULL;
}
```

Main

- start threads … let then run
- Perfect numbers are rare and not related to each other so this amount fo communication and control works
- because perfect numbers are unrelated could easily do process parallel
- Advantages of thread parallel for this task over process parallel?

```
int main(int argc, char const *argv[])
{
    pthread_mutex_init(&mutexAV, NULL);

    perfect.count = 0;
    sprintf(perfect.header, "perfect");

    N = atoi(argv[1]);
    curr = 0;
    pthread_t threads[NTHREADS];
    threadInfo tInfo[NTHREADS];
    for (int i = 0; i < NTHREADS; i++)
    {
        tInfo[i] = (threadInfo){.threadNum = i};
        pthread_create(threads + i, NULL, checkit, &tInfo[i
    }
    for (int i = 0; i < NTHREADS; i++) {
        pthread_join(threads[i], NULL);
        printf("Joined %d\n", i);
    }

    pres(&perfect);
   return 0;
```

Finding Primes

- Again using a brute force approach
    - check if number is evenly divisible by all smaller primes

- Problem, need to do a lot more with the recording than with perfect numbers
    - there are a lot more primes than perfect numbers
- BUT the code is largely the same

primes

Only odds

Get a local copy of this value. Otherwise can get problems when another thread updates. Could handle with atomic or locks. This is faster and works for this problem

```
void* getPrimes(void * args) {
    threadInfo *tInfo = args;
    while (primes.count < MAXX - 1)
    {
        long np = atomic_fetch_add(&num, 2);
        long sq = sqrt(np);
        int g = 1; // true a long as he number could still be prime
        int mxx = primes.count - 2;
        for (long i = 0; g == 1 &&
                         i < mxx &&
                         primes.data[i] <= np; i++)
        {
            if (np % primes.data[i] == 0)
                g = 0;
        }
        if (g) {
            updatePrimes(np);
            if (primes.count % 1000 == 0)
        printf("%d %d %ld\n", tInfo->threadNum, primes.count, primes.
        }}}
```

Other forms of Parallelism

- My examples were both of compute bound problems so 1 thread (or process) per core is entirely reasonable.
  - multiple thread — single memory
- For an I/O bound process other choices
  - maybe many more threads than cores as most threads are idle
    - still might be multiple thread — single memory
  - E.g. Web Browser

Lab

- Find evidence that your browser really does things in parallel.
    - Hint, in chrome, look in Developer / Developer Tools
    - You can definitely do this without using developer tools, but you are a developer so …
- Send a screenshot (or picture) of something that shows parallel actions with a brief paragraph of how your screenshot shows parallel.