Concurrent Programming

What is Concurrent Programming?

I'm trying to come up right now with an example of a common device that doesn't use modern multi-core processors...uhm, phones...no...tablets...no...Ipod touch...no, smartwatches...no -at least not the big-league ones; Fitbit ones do, so do as well any ipod nano...

I think you get the idea though: if it looks toylike, it has a single-core processor; otherwise, it's multi-core.

In this post I want to address the following question at a introductory level:

> **How can we take advantage in our code of those multiple cores**?
> **Answer**: By having our program spawn multiple children processes that may do the main calculation and letting the parent process collect those results.
> In other words, **by using concurrent programming**!

A basic way of achieving this is using the fork operation available in all *NIXs (OSX, Linux, FreeBSD...).

A different way of taking advantage of multiple cores is through the use of threads, e.g, pthreads. In linux there is no difference -see the reference by
Linux Torvald further down; in Windows, there is no fork, only threads.

In Linux there is even a third way, closely related to fork, but that gives you a much more fine-grained level of control: clone and clone3. See the man pages.
It is this call, in either version, clone/clone3, in Linux that really blurs the historical distinction between process and threads
making both just but two ends of a spectrum of possible concurrent programming models. All three calls are part of the same family of system calls.
This family includes as well other calls.

Index:

The Multi-Process / Parallel / Concurrent "Hello World!" ^

Let's see an example in C:

```
//helloworld-mp.c

#include <stdio.h>      //printf
#include <unistd.h>     //fork
#include <stdlib.h>     //exit

int main(){
    pid_t pid = fork();

    switch( pid ) {
        case -1:
            printf("ERROR: spawning of child process failed\n");
            return 1;
            break;
        case 0:
            printf("Hello World! I'm the child whose process id pid=%d...
Uhm...odd, yeah...\n"\
                    "anyway, my parents know my pid for sure. Ask
them!\n",pid);
            exit(0);
            break;
        default:
            printf("Hi, I'm the parent of the process with
pid=%d\n",pid);
            break;
    }
```

```
        return 0;
    }
```

Compile as `gcc helloworld-mp.c` and run as `./a.out`

The output should look like this

> Hi, I'm the parent of the process whose pid=11518
> Hello World! I'm the child with process id pid=0... Uhm...odd, yeah...
> anyway, my parents know my pid for sure. Ask them!

What just happened? ^

The call to fork basically generates a copy of the whole program and its state up to that point -except for file descriptors, which are shared, and the value of the pid variable, which differs:

- During the program execution
  - the child, any child, sees a value of pid == 0.
  - The parent process sees a pid value that varies from child to child. That pid value seen by the parent is the actual PID the kernel assigned to the child process.
- The memory pages of our program are likely duplicated. I say likely because the method used is **copy-on-write (COW)**. Basically this works as follows:
  - initially things aren't duplicated at all (except details like the pid variable, say). The child process has access to any variable as would do any normal code of ours.
  - However, **if and when the child (or parent) tries to modify a common variable, the kernel copies that variable to the child (or parent) memory page**.
- The children are assigned a PGRP (group process id) equal to the PID of the parent.

As the virtual pages of different processes are independent, by default those modifications done by the child process cannot be seen by the parent; and vice-versa.

You can find more details at this excellent stackoverflow thread, or this comment of Linus Torvald on the way Linux deals with processes and threads.

Non-determinism ^

In concurrent programing execution is in general non-deterministic. This means, we cannot anticipate the order in which the different child process will be processed and, hence, finish their tasks. Thus, you may have seen the above output with the child's greeting first. That's more likely so the more children your program spawns and the more busy your computer is running apps in the background -browsers with several tabs, mail program, itunes...

This fact is the source of additional headaches when debugging a concurrent program.

Multiple Children ^

Just do that same fork as many times as you need. Here we will use a loop.

```
#include <stdio.h>      //printf
#include <unistd.h>     //fork
#include <stdlib.h>     //exit

#define NUMBER_CHILDREN 8

int main(){
    for ( int i= 0 ; i < NUMBER_CHILDREN ; i++){
        pid_t pid = fork();

        switch( pid ) {
            case -1:
                printf("ERROR: spawning of child process failed\n");
                return 1;
                break;
            case 0:
                printf("Hello World! I'm the child number %d. Bye!\n",i+1);
                exit(0);
                break;
            default:
                printf("Hi, I'm the parent of the process with 
pid=%d\n",pid);
                break;
        }
    }
    return 0;
}
```

Compile and run. You should see an output similar to this:

```
Hi, I'm the parent of the process with pid=11607
Hi, I'm the parent of the process with pid=11608
Hello World! I'm the child number 1. Bye!
Hello World! I'm the child number 2. Bye!
Hi, I'm the parent of the process with pid=11609
Hello World! I'm the child number 3. Bye!
Hi, I'm the parent of the process with pid=11610
```

```
Hello World! I'm the child number 4. Bye!
Hi, I'm the parent of the process with pid=11611
Hello World! I'm the child number 5. Bye!
Hi, I'm the parent of the process with pid=11612
Hello World! I'm the child number 6. Bye!
Hi, I'm the parent of the process with pid=11613
Hello World! I'm the child number 7. Bye!
Hi, I'm the parent of the process with pid=11614
Hello World! I'm the child number 8. Bye!
```

Run it several times. You'll see what I meant by non-determinism: The order in which those print statements are executed varies from run to run.

Chaperoning your children ^

Any decent parents will want to look after their children. I know you do. The child's PID allows the parent some level of targeted interaction, like...well, killing it. 8-/

Another thing that parents often do, is wait for their children to come back after they rushed to the playground and played for a while.

We can do this the following way:

```c
#include <stdio.h>       //printf
#include <unistd.h>      //fork
#include <stdlib.h>      //exit

#define NUMBER_CHILDREN 8

int main(){
    pid_t pid;
    for ( int i= 0 ; i < NUMBER_CHILDREN ; i++){
        pid = fork();

        switch( pid ) {
            case -1:
                printf("ERROR: spawning of child process failed\n");
                return 1;
                break;
            case 0:
                printf("Hello World! I'm the child number %d. Bye!\n",i+1);
                exit(0);
```

```c
                break;
            default:
                printf("Hi, I'm the parent of the process with
pid=%d\n",pid);
                break;
        }
    }
    printf("I'm the parent waiting for all my %d offspring to come back and
report\n",NUMBER_CHILDREN);
    int status;
    while( (pid=wait(&status)) > 0) {
        printf("My child with pid=%d is back!\n",pid);
    }
    printf("All children returned\n");
    return 0;
}
```

Compile as usual and run it. You should see something like this:

```
Hi, I'm the parent of the process with pid=11685
Hi, I'm the parent of the process with pid=11686
Hello World! I'm the child number 1. Bye!
Hello World! I'm the child number 2. Bye!
Hi, I'm the parent of the process with pid=11687
Hi, I'm the parent of the process with pid=11688
Hello World! I'm the child number 3. Bye!
Hello World! I'm the child number 4. Bye!
Hi, I'm the parent of the process with pid=11689
Hi, I'm the parent of the process with pid=11690
Hello World! I'm the child number 5. Bye!
Hello World! I'm the child number 6. Bye!
Hi, I'm the parent of the process with pid=11691
Hi, I'm the parent of the process with pid=11692
I'm the parent waiting for all my 8 offsprings to come back and report
Hello World! I'm the child number 7. Bye!
My child with pid=11690 is back!
My child with pid=11689 is back!
My child with pid=11688 is back!
My child with pid=11687 is back!
My child with pid=11686 is back!
My child with pid=11685 is back!
```

```
Hello World! I'm the child number 8. Bye!
My child with pid=11691 is back!
My child with pid=11692 is back!
All children returned
```

The function `wait(int*)` here is a wrapper around `wait4(pid_t,int*,int, struct rusage*)`, equivalent to `wait4(-1,&status,0,NULL)`. See man 2 wait.

`wait(int*)` basically waits -that is, blocks execution!- until any of the children terminates -or is killed- and then it returns the child PID if a child terminated or -1 if there is no more children.

Actually, it returns -1 as well if there was a problem. But you can catch an error value and double check what happened. I'm trying to keep things as simple as possible to get you started though.

Therefore, by looping while wait returns a positive value effectively we are waiting till each and every child has ended its execution.

The variable status contains information of how the child processes terminated its execution -did it really finished as expected? whas it terminated because it received a (e.g. kill) signal? etc.

What have you done out there my child? ^

The following code `waitforallchild.c` provides a more detailed example on what information can be gathered on the child's execution.

It has the child calculating a large Fibonacci number. The code for this is in the header fibo.h listed afterwards:

```c
//waitforallchild.c

#include <stdio.h>
#include <sys/wait.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>
#include <unistd.h>

#ifdef FIBO
#include "fibo.h"
#ifndef FIBON
#define FIBON (fibo_t) 5*1000*1000*1000
#endif
#define childwork() printf("fib(%llu)=%llu\n",FIBON,fib(FIBON));
#define BUSY "computation lasted"
#else
```

```c
#define childwork()  nanosleep(&timeout,NULL);
#define BUSY "slept for"
#endif

#define NUMBER_CORES 8
#define CHILD_SLEEP 10*NUMBER_CORES
#define MAXCHILDREN 100

int main(int argc, const char** argv){
    time_t begin = time(NULL);
    int nch = MAXCHILDREN/25;

    if( argc > 1 && strncmp(argv[1],"-",1)==0 ){
        printf("Usage: %s [number_of_children (%d)]\n\n"\
            "Spawns number_of_children and waits for them to finish before
wrapping up.\n"\
            "Max number allowed is %d. If number_of_children is larger it
automatically reset to that max value.\n",\
            argv[0], nch,MAXCHILDREN\
        );
        exit(0);
    }
    if (argc > 1){
        nch = atoi(argv[1]);
        nch = (nch<0||nch>MAXCHILDREN)? MAXCHILDREN*(nch>0?10:1)/10 : nch ;
    }

    pid_t* pids = (pid_t*) calloc(nch,sizeof(pid_t));

    struct timespec timeout;
    timeout.tv_sec = CHILD_SLEEP/nch;
    timeout.tv_nsec = (CHILD_SLEEP%nch)*100*1000*1000;
    printf("Will try spawning %d children each living for %4.f
sec\n",nch,timeout.tv_sec+1e-9*timeout.tv_nsec);

    pid_t pid , sumOfpids = 0;
    int i , n=0, status;
    for( i= 0 ; i < nch ; i++){
        pid = fork();
        n++;
```

```
        switch(pid){
            case -1:
                printf("WARNING: Error while forking a child. Will try to
continue with remaining %d children.\n",nch-n);
                n--;
                continue;
                break;
            case 0:
                begin = time(NULL);
                printf("Hi, I'm child %d. Going to sleep now\n",n);
                //printf("fib(%llu)=%llu\n",FIBON,fib(FIBON));
                //nanosleep(&timeout,NULL);
                childwork();
                printf("Child %d %s %tu sec. Bye\n",n,BUSY,time(NULL)-begin);
                exit(0);
                break;
            default:
                sumOfpids += pid;
                printf("Parent: saw child %d (%d) parting\n",n,pid);
                break;
        }

    }

    printf("All children submitted. Waiting for their termination...\n");

    /*
    //all theese are equivalent ways for waiting FOR ALL children -caveat: if
a child stops (but not terminating) waitn returns a >0 value too!
    //while( waitpid(0,&status,0)>0 );
    //while( wait4(-1,&status,0,0)>0 );
    //while( wait3(&status,0,0)>0 );
    //while( wait(&status)>0 );

    while( sumOfpids > 0 ){
        pid = wait(&status);
        if( pid == -1 ){
            printf("Error in child or while waiting for one\n");
            exit(1);
        }
```

```
        sumOfpids -= pid ;
        printf("Child %d finished\n",pid);
    }
    */
    //wait and check exit status
    int options = WUNTRACED;
    while( (pid=wait3(&status,options,0))>0){
        int wexstat;
        if ( WIFEXITED(status) ){
            wexstat=WEXITSTATUS(status);
            printf("Child %d terminated with an exit call %d\n",pid,wexstat);
        }
        if ( WIFSIGNALED(status)) {
            wexstat=WTERMSIG(status);
            printf("Child %d terminated with a term signal
%d\n",pid,wexstat);
            if( WCOREDUMP(status) ) printf("Coredump was created\n");
        }
        if ( WIFSTOPPED(status)) {
            wexstat=WSTOPSIG(status);
            printf("Child %d stopped with signal %d\n",pid,wexstat);
        }
    }
    printf("All %d children finished.Elapsed time aprox. %tu
sec.\n",n,time(NULL)-begin);

    return 0;
}
```

The fibo.h header :

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#ifndef DEBUG
#define DEBUG 0
#endif

typedef long long unsigned int fibo_t;
```

```
fibo_t  fib(fibo_t n){
        fibo_t f, tmp1, tmp2;
        switch(n){
                case 0:
                        return 0;
                        break;
                case 1:
                case 2:
                        return 1;
                        break;
                default:
                        tmp2 = (fibo_t) 1;
                        tmp1 = (fibo_t) 1;
                        for(fibo_t i=2; i < n ; i++){
                                f = tmp1 + tmp2;
                                if(DEBUG)printf("#DEBUG:
fib(%llu)=%llu+%llu=%llu\n",i+1,tmp1, tmp2, f);
                                tmp2 = tmp1 ;
                                tmp1 = f;
                        }
                        return f;
        }
        exit(-1);
}
```

Compile as `gcc -o waitforallchild waitforallchild.c -DFIBO` then run as `./waitforallchild`

You should see something like this:

```
Will try spawning 4 children each living for   20 sec
Parent: saw child 1 (11829) parting
Parent: saw child 2 (11830) parting
Hi, I'm child 1. Going to sleep now
Parent: saw child 3 (11831) parting
Hi, I'm child 2. Going to sleep now
Parent: saw child 4 (11832) parting
All children submitted. Waiting for their termination...
Hi, I'm child 3. Going to sleep now
Hi, I'm child 4. Going to sleep now
fib(5000000000)=10859268830657044933
Child 1 computation lasted 20 sec. Bye
```

```
Child 11829 terminated with an exit call 0
fib(5000000000)=10859268830657044933
Child 2 computation lasted 20 sec. Bye
Child 11830 terminated with an exit call 0
fib(5000000000)=10859268830657044933
Child 3 computation lasted 20 sec. Bye
Child 11831 terminated with an exit call 0
fib(5000000000)=10859268830657044933
Child 4 computation lasted 20 sec. Bye
Child 11832 terminated with an exit call 0
All 4 children finished.Elapsed time aprox. 20 sec.
```

Now open a new terminal window. We will run it again but this time kill one of the children while still computing the fibonacci number. In this example I issued a `kill -9 11837` and the output is:

```
Parent: saw child 1 (11836) parting
Hi, I'm child 1. Going to sleep now
Parent: saw child 2 (11837) parting
Hi, I'm child 2. Going to sleep now
Parent: saw child 3 (11838) parting
Parent: saw child 4 (11839) parting
All children submitted. Waiting for their termination...
Hi, I'm child 3. Going to sleep now
Hi, I'm child 4. Going to sleep now
Child 11837 terminated with a term signal  9
fib(5000000000)=10859268830657044933
Child 4 computation lasted 20 sec. Bye
Child 11839 terminated with an exit call 0
fib(5000000000)=10859268830657044933
Child 3 computation lasted 20 sec. Bye
Child 11838 terminated with an exit call 0
fib(5000000000)=10859268830657044933
Child 1 computation lasted 20 sec. Bye
Child 11836 terminated with an exit call 0
All 4 children finished.Elapsed time aprox. 20 sec.
```

Matrix Multiplication ^

Multi-linear functions, aka. Matrices are a domain of Linear Algebra that can easily benefit from using multiple cores.

One example is matrix multiplication.

Let's denote a matrix A with n rows and m columns as A(n,m).

A matrix A(n,k) can only be multiplied (on the right) with a matrix B(k,m). The resulting matrix will have n rows and m columns. Lets denote it as C(n,m).

Thus C has n*m components. Each of these can be calculated independently as follows: row i, column j of C is given by

C(i,j) = product, element by element, of row i of A times column j of B and then adding all those k results = A(i,1)*B(1,j) + A(i,2)*B(2,j) + ... + A(k,1)*B(k,j)

That is, the determination of each element of C involves 2k-1 operations: k multiplications plus k-1 addtions. Hence, in total the matrix multiplication requires n*m*(2k-1) operations.

In particular, calculating the square of a matrix A(n,n) involves n*n*(2n-1) ~ 2n^3 operations.

The following code example does this multi-threaded, or rather, multi-process matrix multiplication. It relies on the header `mtxmult_mp.h` shown afterwards.

```
//test_mtxmult_mp-big.c

#include "mtxmult_mp.h"
#include <math.h>
#include <string.h>

int main(int argc, char** argv){
    size_t nbproc = NUM_PROCESS;
    if( argc < 3){
        printf("Usage: %s nrows ncols [ nbproc (%zu) ] [-ns]\n\n-ns\t\tno 
single-thread\n",\
            argv[0],nbproc);
        return -1;
    }
    size_t nrows = (size_t) atol(argv[1]);
    size_t ncols = (size_t) atol(argv[2]);
    if( argc > 3 ){
        nbproc = (size_t) atol(argv[3]);
    }
    int skip_single_thread=0;
    if ( argc > 4 && strcmp("-ns",argv[4])==0){
        skip_single_thread=1;
    }
    printf("Max # threads: %zu\n",nbproc);
    printf("Random array size %zux%zu\n",nrows,ncols);
```

```
    //orig matrix in shared memory makes no difference due to COW and being
    //used just for reading by children
    //double* A = mmap(NULL,nrows*ncols*sizeof(double),PROT_READ |
PROT_WRITE,
    //                                        MAP_SHARED | MAP_ANONYMOUS, -1,
0);
    double *A2, *A = (double*) calloc(nrows*ncols, sizeof(double));
    srand(1234567);
    for(size_t i = 0 ; i < nrows*ncols ; i++){
         A[i] = floor((10.0*rand())/RAND_MAX) ;
    }
    const size_t asz[2]={nrows,ncols};
    pmtx(A,nrows,ncols,"A");
    time_t begin ;
    if( !skip_single_thread ){
        printf("Single thread\n");
        begin = time(NULL);
        //pmtx( mtxsq(A,asz),nrows,ncols,"A^2");
        A2=mtxsq(A,asz);
        printf("Time: %zu sec.\n",time(NULL)-begin);
        pmtx(A2,nrows,ncols,"A^2");
    }
    begin = time(NULL);
    //pmtx( mtxsq_thr(A,asz,nbproc),nrows,ncols,"A^2");
    A2=mtxsq_thr(A,asz,nbproc);
    printf("Time: %zu sec.\n",time(NULL)-begin);
    pmtx(A2,nrows,ncols,"A^2");
    return 0;
}
```

The actual concurrent calculation of the matrix multiplication is done by the code in `mtxmult_mp.h` that follows. As this example intends to show the difference between concurrent and sequential calculation of the square of a matrix, this code in turn requires the (sequential) version found in the header `mtxmult.h` that is listed afterwards.

```
//mtxmult_mp.h

#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <unistd.h>
```

```c
#include <time.h>
#include <sys/mman.h>

#include "mtxmult.h"

#define NUM_CORES 4
#define NUM_PROCESS (size_t)(NUM_CORES-1)
#define DEBUG 0

//
//Threaded ( via fork() ) version of matrix multiplication

double* mtxm_thr(double* a,const size_t asz[2],double* b,const size_t bsz[2],
size_t nbproc){
    printf("Threaded Matrix Multiplication start...%zu threads\n",nbproc);
    if ( asz[1] != bsz[0] ){
        printf("ERROR: can't multiply A*B: columns of A %zu != rows of
b%zu\n",asz[1],bsz[0]);
        exit(1);
    }
    //result matrix in shared memory
    double* ab = mmap(NULL,asz[0]*asz[1]*sizeof(double),PROT_READ |
PROT_WRITE,
                                                   MAP_SHARED | MAP_ANONYMOUS, -1,
0);
    if( ab == MAP_FAILED){
        printf("ERROR: Couldn't allocate shared memory for matrix
multiplication");
        exit(2);
    }

    size_t i,j,k, mop, mopo, pid, children=0, fchildren=0;
    size_t mpt = (asz[0]*bsz[1])/nbproc;
    size_t rmm = (size_t) ((asz[0]*bsz[1])%nbproc);
    if( mpt == 0){
        nbproc=asz[0]*bsz[1];
        mpt = 1;
        rmm = 0;
        printf("I: Too large number of processes. Reset to %zu\n",nbproc);
    }
```

```
    size_t* fchildren_ptr = &fchildren;
    mopo=rmm;
    int pd[2];
    pipe(pd);

    while ( children < nbproc - 1 ) {
        children++;
        mopo += mpt;
        pid = fork();
        if ( !pid ){
            for(mop= mopo ; mop < (mopo+mpt) ; mop++){//
                i = (size_t) (mop/bsz[1]);
                j = (mop%bsz[1]);
                for(k=0; k<asz[0] ; k++){
                    ab[i*asz[1]+j] += a[i*asz[1]+k] * b[k*bsz[1]+j] ;
                }
                if( DEBUG ) printf("#DEBUG: child %zu :
ab(%zu,%zu)=%.4f\n",children,i,j,ab[i*asz[1]+j]);
            }
            (*fchildren_ptr)++;

            if( write(pd[1], fchildren_ptr, sizeof(size_t)) == -1 ){
                printf("ERROR: child %zu couldn't write to
parent\n",children);
                exit(3);
            }
            //return 0;
            exit(0);
        }
    }
    if( DEBUG ) printf("#DEBUG: Parent processing... mop[0, %zu)\n",rmm+mpt);
    for(mop=0 ; mop < rmm+mpt ; mop++){
        i = (size_t) (mop/bsz[1]);
        j = (mop%bsz[1]);
        for(k=0; k<asz[0] ; k++)
            ab[i*asz[1]+j] += a[i*asz[1]+k] * b[k*bsz[1]+j] ;
        if( DEBUG ) printf("#DEBUG: parent :
ab(%zu,%zu)=%.4f\n",i,j,ab[i*asz[1]+j]);
    }
```

```
    struct timespec timeout;
    timeout.tv_sec = 1;
    timeout.tv_nsec = 10*1000;

    while( fchildren< nbproc-1 ){
        if( DEBUG ) printf("#DEBUG: checking... %zu(%zu) children
finished\n",fchildren,nbproc-1);
        nanosleep(&timeout,NULL);
        if( read(pd[0],&children,sizeof(size_t)) == -1 ){
            printf("ERROR: parent :  reading pipe \n");
            exit(4);
        }
        fchildren += children;
    }
    if( DEBUG ) printf("#DEBUG: %zu(%zu) children
finished\n",fchildren,nbproc-1);
    return ab;
}
double* mtxsq_thr(double* m, const size_t msz[2], size_t nbproc){
    return mtxm_thr(m,msz,m,msz,nbproc);
}
```

And finally here the linear matrix multiplication code of `mtxmult.h`

```
//mtxmult.h
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>
void pmtx(double* m, size_t rs, size_t cs,char* txt){
    if( rs*cs <= 0 ) {
        fprintf(stderr,"Warning: ill-defined matrix dimensions\n");
        return;
    }
    if( txt ) printf("%s\n",txt);
    size_t i,j ;
    for(i=0 ; i<rs ; i++){
        for(j=0 ; j<cs ; j++){
            printf("%.4f\t",m[i*cs+j]);
            if( j == 10) {
```

```
                j = cs;
                printf("...");
            }
        }
        printf("\n");
        if( i == 10){
            i = rs;
            printf("...\n");
        }
    }
}
double* mtxm(double* a,const size_t asz[2],double* b,const size_t bsz[2]){
    if ( asz[1] != bsz[0] ){
        printf("ERROR: can't multiply A*B: columns of A %zu != rows of
b%zu\n",asz[1],bsz[0]);
        exit(1);
    }
    double* ab = calloc(asz[0]*bsz[1],sizeof(double));
    if( !ab ){
        printf("ERROR: Couldn't allocate memory for matrix multiplication");
        exit(2);
    }
    size_t i,j,k;
    for(i=0 ; i<asz[0] ; i++){
        for(j=0 ; j<bsz[1] ; j++)
            for(k=0; k<asz[0] ; k++)
                ab[i*asz[1]+j] += a[i*asz[1]+k] * b[k*bsz[1]+j] ;
    }
    return ab;
}
double* mtxsq(double* m, const size_t msz[2]){
    return mtxm(m,msz,m,msz);
}
```

In a Mac, compile this simply as `gcc -o test_mtxmult_mp-big test_mtxmult_mp-big.c`.

In Linux you'll likely have to hint the linker to the right math library with -lm, thus compile it as `gcc -o test_mtxmult_mp-big test_mtxmult_mp-big.c -lm`.

Do a first test by running `./test_mtxmult_mp-big 3 3` You should get

```
Max # threads: 3
Random array size 3x3
A
6.0000    2.0000    3.0000
3.0000    8.0000    7.0000
7.0000    9.0000    9.0000
Single thread
Time: 0 sec.
A^2
63.0000    55.0000    59.0000
91.0000    133.0000    128.0000
132.0000    167.0000    165.0000
Threaded Matrix Multiplication start...3 threads
Time: 2 sec.
A^2
63.0000    55.0000    59.0000
91.0000    133.0000    128.0000
132.0000    167.0000    165.0000
```

You can see that both calculations of the square of matrix A coincide. That's what we want to see.

But you may also notice that the multi-process calculation takes slightly more to complete!

There is an overhead the kernel incurs in setting up everything for us to use multiple processes. For small matrices that's actually a noticeable burden and a single process fares better.

When is a matrix big enough to justify the use of concurrent programing? ^

The above program `test_mtxmult_mp-big` creates a random square matrix of the size we specify in the command line and calculates its square both sequentiall and concurrently with as many child processes as specified. Notice that the parent also takes up its share of work. Thus using 7 children actually means splitting the work among 8 concurrent processes that do a calculation!

Suprisingly, it takes quite a large matrix to start seeing a benefit. The actual size will depend on your machine's specs.

In this first result I'm using a MacbookPro with the following specs

```
$system_profiler SPHardwareDataType

      Model Name: MacBook Pro
      Model Identifier: MacBookPro9,1
      Processor Name: Quad-Core Intel Core i7
```

```
Processor Speed: 2.3 GHz
Number of Processors: 1
Total Number of Cores: 4
L2 Cache (per Core): 256 KB
L3 Cache: 6 MB
Hyper-Threading Technology: Enabled
Memory: 16 GB
Boot ROM Version: 233.0.0.0.0
SMC Version (system): 2.1f173
Serial Number (system): CH348DJ39J4
Hardware UUID: 723078B1-962D-3749-8B77-1D1D857678B1
Sudden Motion Sensor:
State: Enabled
```

```
$sysctl -n machdep.cpu.brand_string
Intel(R) Core(TM) i7-3615QM CPU @ 2.30GHz
```

This table summarizes the results

Size | Single thread | MP 8 threads
1000 | 11 sec | 10 sec
3000 | 400 sec | 107 sec

Thus up to a 1000*1000 matrix it isn't worth bothering with multi-processing. That's roughly 2 billion operations where a single cores competes well against 8. Impressive.

For a matrix of size 3000*3000 the difference is however clear: MP using 8 processes has a clear advantage by being roughly 4 times faster!

```
For this latter case, the following is a snapshot of resource usage as seen by top:
Processes: 455 total, 10 running, 445 sleeping, 1897 threads
01:55:23
Load Avg: 5.47, 2.85, 2.42  CPU usage: 97.98% user, 1.88% sys, 0.12% idle
SharedLibs: 315M resident, 53M data, 38M linkedit.
MemRegions: 103199 total, 3978M resident, 128M private, 1540M shared.
PhysMem: 10G used (2072M wired), 5932M unused.
VM: 2610G vsize, 1994M framework vsize, 25690086(0) swapins, 26376915(0) swapouts.
Networks: packets: 10311058/10G in, 10417725/8575M out.
Disks: 5873635/201G read, 3603239/189G written.

PID    COMMAND      %CPU  TIME     #TH   #WQ   #PORT MEM    PURG   CMPRS PPID
```

```
PGRP  STATE
12061  test_mtxmult 98.4  00:04.28 1/1  0    8     69M+   0B    0B    12056
12056 running
12058  test_mtxmult 97.5  00:04.28 1/1  0    8     69M+   0B    0B    12056
12056 running
12056  test_mtxmult 97.1  00:04.35 1/1  0    10    69M+   0B    0B      490
12056 running
12057  test_mtxmult 96.5  00:04.24 1/1  0    8     69M+   0B    0B    12056
12056 running
12063  test_mtxmult 96.4  00:04.25 1/1  0    8     69M+   0B    0B    12056
12056 running
12060  test_mtxmult 96.4  00:04.28 1/1  0    8     69M+   0B    0B    12056
12056 running
12059  test_mtxmult 96.4  00:04.25 1/1  0    8     69M+   0B    0B    12056
12056 running
12062  test_mtxmult 96.3  00:04.25 1/1  0    8     69M+   0B    0B    12056
12056 running
```

Notice what we mentioned above:

- The parent process (highlighted in red) has a PID 12056 and a PPID of 490.
- All children have a group process id PGRP (highlighted in green) equal to the PID of the parent, i.e., 1206.

We store the original matrix with 9*10^6 doubles (8 Bytes), that is, about 70MiB . Top reports 69MiB.

1T (trillion) Operations^

In another box I have a Linux system with the following arch (issue lscpu)

```
Architecture:                    x86_64
CPU op-mode(s):                  32-bit, 64-bit
Byte Order:                      Little Endian
Address sizes:                   36 bits physical, 48 bits virtual
CPU(s):                          8
On-line CPU(s) list:             0-7
Thread(s) per core:              2
Core(s) per socket:              4
Socket(s):                       1
NUMA node(s):                    1
Vendor ID:                       GenuineIntel
CPU family:                      6
Model:                           58
```

```
Model name:                      Intel(R) Core(TM) i7-3770 CPU @ 3.40GHz
Stepping:                        9
CPU MHz:                         1596.518
CPU max MHz:                     3900.0000
CPU min MHz:                     1600.0000
BogoMIPS:                        6787.40
Virtualization:                  VT-x
L1d cache:                       128 KiB
L1i cache:                       128 KiB
L2 cache:                        1 MiB
L3 cache:                        8 MiB
NUMA node0 CPU(s):               0-7
Vulnerability Itlb multihit:     KVM: Mitigation: Split huge pages
Vulnerability L1tf:              Mitigation; PTE Inversion; VMX conditional
cache flushes, SMT vulnerable
Vulnerability Mds:               Vulnerable: Clear CPU buffers attempted, no
microcode; SMT vulnerable
Vulnerability Meltdown:          Mitigation; PTI
Vulnerability Spec store bypass: Vulnerable
Vulnerability Spectre v1:        Mitigation; usercopy/swapgs barriers and
__user pointer sanitization
Vulnerability Spectre v2:        Mitigation; Full generic retpoline, STIBP
disabled, RSB filling
Vulnerability Tsx async abort:   Not affected
Flags:                           fpu vme de pse tsc msr pae mce cx8 apic sep
mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe
syscall nx rdtscp lm constant_tsc arch_perfmon pebs bts rep_good nopl xtopology
nonstop_tsc cpuid aperfmperf pni pclmulqdq dtes64 monitor ds_cpl vmx smx est
tm2 ssse3 cx16 xtpr pdcm pcid sse4_1 sse4_2 x2apic popcnt tsc_deadline_timer
aes xsave avx f16c rdrand lahf_lm cpuid_fault epb pti tpr_shadow vnmi
flexpriority ept vpid fsgsbase smep erms xsaveopt dtherm ida arat pln pts
```

Calculations will be faster here simply due to the 1.5 times faster CPU clock.

```
Size   |  Single thread | MP 8 threads
1000  |         10 sec     |          9 sec
3000  |       341 sec     |        84 sec
10000|   15972 sec     |    7101 sec = 118min (w/o single-proc calc: 7040 sec =
117.3 min)
```

15972 sec = 4.44hrs or 4h 25min ! More than double the time required by our multitasking version!

I didn't want to leave my laptop running +4h. That's why I didn't provide the timing for the case of a 10,000*10,000 matrix. That's 100M elements requiring a total 1T (trillion or 1e9) operations!

As the elements are stored as doubles, such a matrix requires of the order of 10^8*8Bytes = 800MB = 763MiB. From the previous results we'd expect the single process to use up to ~1.5GiB and take ~10,000sec ~ 2.8hrs, likely though +3hrs, to complete.

A single thread starts occupying around 763MiB. Memory usage grows along the calculation though.
After an 1h 20min it's almost 1GiB:

```
top - 04:05:02 up 35 days,  5:18,  3 users,  load average: 1.11, 1.11, 1.04
Tasks: 216 total,   2 running, 214 sleeping,   0 stopped,   0 zombie
%Cpu(s): 12.5 us,  0.0 sy,  0.0 ni, 87.4 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
MiB Mem :  15808.6 total,    695.3 free,   1960.8 used,  13152.5 buff/cache
MiB Swap:      0.0 total,      0.0 free,      0.0 used.  13090.2 avail Mem

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
 256940 admin     20   0 1566132 990.4m   1736 R  99.9   6.3  78:35.40
test_mtxmult_mp
```

After over 3hrs the memory footprint is 3 times larger

```
top - 06:00:41 up 35 days,  7:14,  3 users,  load average: 1.00, 1.00, 1.01
Tasks: 216 total,   2 running, 214 sleeping,   0 stopped,   0 zombie
%Cpu(s): 12.5 us,  0.0 sy,  0.0 ni, 87.4 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
MiB Mem :  15808.6 total,    364.8 free,   2291.2 used,  13152.6 buff/cache
MiB Swap:      0.0 total,      0.0 free,      0.0 used.  12759.7 avail Mem

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
 256940 admin     20   0 1566132   1.3g   1736 R  99.8   8.3 194:05.29
test_mtxmult_mp
```

Even further, after 4hrs

```
top - 14:17:57 up 35 days, 15:31,  3 users,  load average: 1.00, 1.00, 1.00
Tasks: 218 total,   2 running, 216 sleeping,   0 stopped,   0 zombie
%Cpu(s): 12.5 us,  0.0 sy,  0.0 ni, 87.4 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
MiB Mem :  15808.6 total,    225.9 free,   2428.4 used,  13154.4 buff/cache
MiB Swap:      0.0 total,      0.0 free,      0.0 used.  12622.6 avail Mem

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
 259259 admin     20   0 1566132   1.4g   1700 R  99.9   9.2 240:55.60
test_mtxmult_mp
```

Somewhat later it has reached the threshold of 1.5GiB

```
top - 14:29:57 up 35 days, 15:43,  2 users,  load average: 1.25, 1.17, 1.08
Tasks: 213 total,   3 running, 210 sleeping,   0 stopped,   0 zombie
%Cpu(s): 12.5 us,  0.0 sy,  0.0 ni, 87.4 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
MiB Mem :  15808.6 total,    196.4 free,   2458.1 used,  13154.1 buff/cache
MiB Swap:      0.0 total,      0.0 free,      0.0 used.  12592.9 avail Mem

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
 259259 admin     20   0 1566132   1.5g   1700 R  99.9   9.4 252:54.93
test_mtxmult_mp
```

Finally, the single-core computation finishes and the multi-core one fires in

```
top - 14:49:41 up 35 days, 16:03,  2 users,  load average: 8.03, 6.22, 3.50
Tasks: 220 total,   9 running, 211 sleeping,   0 stopped,   0 zombie
%Cpu(s): 99.6 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.3 hi,  0.0 si,  0.0 st
MiB Mem :  15808.6 total,    159.3 free,   2518.2 used,  13131.1 buff/cache
MiB Swap:      0.0 total,      0.0 free,      0.0 used.  12488.5 avail Mem

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
 260600 admin     20   0 2347384   1.5g   6968 R  99.7   9.7   6:46.09
test_mtxmult_mp
 259259 admin     20   0 2347384   1.5g   8272 R  99.7   9.7 272:35.76
test_mtxmult_mp
 260597 admin     20   0 2347384   1.5g   4592 R  99.6   9.7   6:45.67
test_mtxmult_mp
 260601 admin     20   0 2347384   1.5g   4064 R  99.6   9.7   6:45.89
test_mtxmult_mp
 260603 admin     20   0 2347384   1.5g   7232 R  99.6   9.7   6:45.78
test_mtxmult_mp
 260599 admin     20   0 2347384   1.5g   6968 R  99.6   9.7   6:45.71
test_mtxmult_mp
 260598 admin     20   0 2347384   1.5g   4328 R  99.4   9.7   6:44.97
test_mtxmult_mp
 260602 admin     20   0 2347384   1.5g   4328 R  99.3   9.7   6:45.12
test_mtxmult_mp
```

Notice how the RES column shows now a huge value right from the start of this new, multi-threaded computation. Remember the fork spawns a clone of our program at the point of the call. As we just did the single-core computation, the parent had allocated 1.5GiB and it is a snapshot of that what's clone. Hence, this value.

See a more detailed account of memory usage

```
top - 15:06:15 up 35 days, 16:19,  3 users,  load average: 8.00, 7.95, 6.50
Tasks: 225 total,  10 running, 215 sleeping,   0 stopped,   0 zombie
%Cpu(s): 99.6 us,  0.1 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.3 hi,  0.1 si,  0.0 st
MiB Mem :  15808.6 total,    152.3 free,   2523.2 used,  13133.1 buff/cache
MiB Swap:      0.0 total,      0.0 free,      0.0 used.  12375.0 avail Mem

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+   CODE    DATA   USED   RSan   RSfd   RSlk   RSsh COMMAND
 259259 admin     20   0 2347384   1.5g  22784 R  99.7   9.8 289:05.95      4 1562832   1.5g   1.5g   1756      0  21028 test_mtxmult_mp
 260600 admin     20   0 2347384   1.5g  23072 R  99.7   9.8  23:16.69      4 1562832   1.5g   1.5g    132      0  22940 test_mtxmult_mp
 260601 admin     20   0 2347384   1.5g  17000 R  99.7   9.8  23:15.88      4 1562832   1.5g   1.5g    132      0  16868 test_mtxmult_mp
 260602 admin     20   0 2347384   1.5g  18056 R  99.7   9.8  23:14.43      4 1562832   1.5g   1.5g    132      0  17924 test_mtxmult_mp
 260599 admin     20   0 2347384   1.5g  22280 R  99.6   9.8  23:15.41      4 1562832   1.5g   1.5g    132      0  22148 test_mtxmult_mp
 260597 admin     20   0 2347384   1.5g  17528 R  99.3   9.8  23:14.97      4 1562832   1.5g   1.5g    132      0  17396 test_mtxmult_mp
 260598 admin     20   0 2347384   1.5g  16208 R  99.3   9.8  23:12.49      4 1562832   1.5g   1.5g    132      0  16076 test_mtxmult_mp
 260603 admin     20   0 2347384   1.5g  21224 R  99.3   9.8  23:14.60      4 1562832   1.5g   1.5g    132      0  21092 test_mtxmult_mp
```

And a few minutes before finishing the calculation

```
top - 16:32:45 up 35 days, 17:46,  3 users,  load average: 8.03, 8.05, 8.02
Tasks: 226 total,  10 running, 216 sleeping,   0 stopped,   0 zombie
%Cpu(s): 99.5 us,  0.1 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.3 hi,  0.1 si,  0.0 st
MiB Mem :  15808.6 total,    156.7 free,   2525.9 used,  13126.0 buff/cache
MiB Swap:      0.0 total,      0.0 free,      0.0 used.  11809.7 avail Mem

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+   SWAP   CODE    DATA   USED   RSan   RSfd   RSlk   RSsh COMMAND
 260597 admin     20   0 2347384   1.6g  95936 R 100.0  10.2 109:23.63      0      4 1562832   1.6g   1.5g    132      0  95804 test_mtxmult_mp
 260599 admin     20   0 2347384   1.6g  90920 R 100.0  10.2 109:20.37      0      4 1562832   1.6g   1.5g    132      0  90788 test_mtxmult_mp
 260602 admin     20   0 2347384   1.6g  87224 R 100.0  10.2 109:19.90      0      4 1562832   1.6g   1.5g    132      0  87092 test_mtxmult_mp
 259259 admin     20   0 2347384   1.6g  92744 R  99.7  10.2 375:12.76      0      4 1562832   1.6g   1.5g   1756      0  90988 test_mtxmult_mp
 260598 admin     20   0 2347384   1.6g  94088 R  99.7  10.2 109:18.44      0      4 1562832   1.6g   1.5g    132      0  93956 test_mtxmult_mp
 260601 admin     20   0 2347384   1.6g  84056 R  99.7  10.2 109:21.23      0      4 1562832   1.6g   1.5g    132      0  83924 test_mtxmult_mp
 260603 admin     20   0 2347384   1.6g  91712 R  99.7  10.2 109:19.54      0      4 1562832   1.6g   1.5g    132      0  91580 test_mtxmult_mp
 260600 admin     20   0 2347384   1.6g  97256 R  99.4  10.3 109:25.30      0      4 1562832   1.6g   1.5g    132      0  97124 test_mtxmult_mp
```

Notice here the values of the RSsh column which end up on the scale of ~95MiB, 1/8 of the total size the matrix occupies. See below.

Linux Memory Types^

A comment on Linux Memory Types is in place in other to clarfy a bit what we are reading there.

From the linux top man page:

```
                         Private | Shared
                     1           |          2
  Anonymous  . stack             |
             . malloc()          |
             . brk()/sbrk()      | . POSIX shm*
             . mmap(PRIVATE, ANON) | . mmap(SHARED, ANON)
            -----------------------+----------------------
             . mmap(PRIVATE, fd) | . mmap(SHARED, fd)
 File-backed . pgms/shared libs  |
                     3           |          4
```

```
       The following may help in interpreting process level memory values
displayed as scalable columns and discussed under topic `3a. DESCRIPTIONS of
```

```
Fields'.

          %MEM - simply RES divided by total physical memory
          CODE - the `pgms' portion of quadrant 3
          DATA - the entire quadrant 1 portion of VIRT plus all explicit mmap
file-backed pages of quadrant 3
          RES  - anything occupying physical memory which, beginning with
Linux-4.5, is the sum of the following three fields:
                 RSan - quadrant 1 pages, which include any former quadrant 3
pages if modified
                 RSfd - quadrant 3 and quadrant 4 pages
                 RSsh - quadrant 2 pages
          RSlk - subset of RES which cannot be swapped out (any quadrant)
          SHR  - subset of RES (excludes 1, includes all 2 & 4, some 3)
          SWAP - potentially any quadrant except 4
          USED - simply the sum of RES and SWAP
          VIRT - everything in-use and/or reserved (all quadrants)

       Note: Even though program images and shared libraries are considered
private to a process, they will be accounted for as shared (SHR) by the kernel.
```

Thus it is

USED = RES + SWAP (0 here; not shown above)

RES = RSan + RSfd + RSsh

SWAP is that chunk of the process' memory that has been moved to a file because of too high a volume of memory requests by other tasks running in your computer.

RSan = Resident Anonymous Memory Size (KiB). Chunk of physical memory allocated to the process that is not linked to any file -that is, it's pure in RAM. This is a private (for the process'-own-eyes only) chunk of memory.

RSfd = Resident File-Backed Memory Size (KiB). Shared pages supporting program images and shared libraries. Also explicit file mappings both private and shared.

RSsh = Resident Shared Memory Size (KiB). Explicitly shared anonymous shm*/mmap pages.

Then we also have

CODE = Chunk (KiB) of physical memory currently used by the executable code, aka. Text Resisdent Set (TRS).

DATA = Data + Stack size (KiB). Memory reserved by the process, aka. Data Resident Set (DRS). It may not yet be mapped to physical (RES) memory. It's always included in VIRT though.

Simple Improvement^

Therefore, the way we wrote our program fork is efectively cloning the memory of the parent process. That's on us.

While at fork time 1.5GiB are cloned and allocated into res memory to each child (RSan) its value stays constant throughout the whole calculation.

However, the USED value increases by roughly 100MiB. This corresponds to the increase in RSsh the shared anonymous resident memory allocated to each child. This value corresponds indeed to the size of the corresponding chunk of the end result matrix each child must compute.

As we allocated a shared memory for the resulting matrix (check the code for double* ab = mmap(...)), the space for the result doesn't amount to an additional 760MiB for each process.
One way to optimize this calculation is to free the space occupied by the result of the single-process calculation. That would leave only the original matrix for cloning, ~760MiB.

Furthermore, we could avoid duplicating this data also. The way would be defining it as shared memory, initialize its values in the parent, and right then, make it read-only. I'll try to post the result another day.

But even without going into that trouble, simply allocating the initial matrix in a shared memory space (mmap) yields already some improvements -instead of a regular calloc (see the code). Check the following two top snapshots.

Both show the initial moments of a run with the 10,000*10,000 matrix and 8 threads, but skipping the single-thread calculation (option -ns).

The first corresponds to allocating the initial matrix via the calloc() system call. It's size of ~760MiB is effectively cloned as anonymous resident memory space, RSan, to each of the threads

```
top - 17:21:00 up 35 days, 18:34,  3 users,  load average: 5.91, 4.47, 2.58
Tasks: 223 total,  10 running, 212 sleeping,   1 stopped,   0 zombie
%Cpu(s): 99.6 us,  0.1 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.3 hi,  0.1 si,  0.0 st
MiB Mem :  15808.6 total,   1678.7 free,   1746.7 used,  12383.2 buff/cache
MiB Swap:      0.0 total,      0.0 free,      0.0 used.  13300.3 avail Mem

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+   SWAP   CODE    DATA   USED   RSan   RSfd   RSlk   RSsh COMMAND
 261508 admin     20   0 1566132 781716    364 R  99.6   4.8   0:35.52      0      4  781580 781716 781352    132      0    232 test_mtxmult_mp
 261509 admin     20   0 1566132 781980    628 R  99.6   4.8   0:35.57      0      4  781580 781980 781352    132      0    496 test_mtxmult_mp
 261510 admin     20   0 1566132 781980    628 R  99.6   4.8   0:35.57      0      4  781580 781980 781352    132      0    496 test_mtxmult_mp
 261514 admin     20   0 1566132 781980    628 R  99.6   4.8   0:35.52      0      4  781580 781980 781352    132      0    496 test_mtxmult_mp
 261512 admin     20   0 1566132 781716    364 R  99.6   4.8   0:35.54      0      4  781580 781716 781352    132      0    232 test_mtxmult_mp
 261513 admin     20   0 1566132 781980    628 R  99.6   4.8   0:35.52      0      4  781580 781980 781352    132      0    496 test_mtxmult_mp
 261507 admin     20   0 1566132 783424   2072 R  99.5   4.8   0:36.68      0      4  781580 783424 781352   1780      0    292 test_mtxmult_mp
 261511 admin     20   0 1566132 781716    364 R  99.1   4.8   0:35.40      0      4  781580 781716 781352    132      0    232 test_mtxmult_mp
```

However, when allocating the initial matrix as shared memory space via the mmap() system call, things look more promising: The whole initial matrix is now allocated as RSsh, while the private RSan chunk is minimal!

```
top - 17:15:42 up 35 days, 18:29,  3 users,  load average: 1.81, 0.41, 0.82
Tasks: 223 total,  10 running, 212 sleeping,   1 stopped,   0 zombie
%Cpu(s): 99.5 us,  0.1 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.3 hi,  0.1 si,  0.0 st
MiB Mem :  15808.6 total,   1677.6 free,    985.8 used,  13145.3 buff/cache
MiB Swap:      0.0 total,      0.0 free,      0.0 used.  13300.6 avail Mem

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+   SWAP   CODE    DATA   USED   RSan   RSfd   RSlk   RSsh COMMAND
 261462 admin     20   0 1566132 781552 781456 R  99.6   4.8   0:15.04      0      4     328 781552     96    132      0 781324 test_mtxmult_mp
 261455 admin     20   0 1566132 783372 783276 R  99.4   4.8   0:16.23      0      4     328 783372     96   1780      0 781496 test_mtxmult_mp
 261456 admin     20   0 1566132 781500 781404 R  99.4   4.8   0:15.03      0      4     328 781500     96    132      0 781272 test_mtxmult_mp
 261458 admin     20   0 1566132 781412 781316 R  99.4   4.8   0:14.98      0      4     328 781412     96    132      0 781184 test_mtxmult_mp
 261460 admin     20   0 1566132 781708 781612 R  99.4   4.8   0:15.02      0      4     328 781708     96    132      0 781480 test_mtxmult_mp
 261461 admin     20   0 1566132 781584 781488 R  99.4   4.8   0:14.99      0      4     328 781584     96    132      0 781356 test_mtxmult_mp
 261459 admin     20   0 1566132 781516 781420 R  99.2   4.8   0:15.02      0      4     328 781516     96    132      0 781288 test_mtxmult_mp
 261457 admin     20   0 1566132 781472 781376 R  99.0   4.8   0:14.98      0      4     328 781472     96    132      0 781244 test_mtxmult_mp
```

COW in Action^

Let's run this: `./test_mtxmult_mp-big 40000 40000 8 -ns`. This is a square matrix of 40,000 rows and columns, hence 16 times larger than the previous example, thus roughly 11.9GiB.

First will do it allocating the initial matrix with a calloc call. That will put it under RSan and Data.

If fork would actually clone and copy this memory for each of the 8 processes, the system would run out of memory, as it only has 16GiB of RAM.

However, when we run it things do work. Furthermore, the physical used memory as given by top is 13368MiB, slightly above 12GiB. This is about 12400MiB more than the baseline (i.e., before starting the calculation). Notice the similarity with the memory required by the matrix itself.

This means that fork didn't really copy and duplicated all that data! It's only being read, so COW should not need to really duplicate -no write.

```
top - 18:47:21 up 35 days, 20:01,  3 users,  load average: 8.29, 39.83, 29.78
Tasks: 224 total,   9 running, 215 sleeping,   0 stopped,   0 zombie
%Cpu(s): 99.7 us,  0.1 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.2 hi,  0.0 si,  0.0 st
MiB Mem :  15808.6 total,    342.7 free,  13368.6 used,   2097.3 buff/cache
MiB Swap:      0.0 total,      0.0 free,      0.0 used.   1697.4 avail Mem

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+  SWAP   CODE   DATA   USED   RSan   RSfd   RSlk   RSsh COMMAND
 262215 admin     20   0   23.8g  11.9g   1680 R  99.7  77.2   0:54.73     0      4  11.9g  11.9g  11.9g   1680      0      0 test_mtxmult_mp
 262220 admin     20   0   23.8g  11.9g      0 R  99.7  77.2   0:36.79     0      4  11.9g  11.9g  11.9g      0      0      0 test_mtxmult_mp
 262221 admin     20   0   23.8g  11.9g      0 R  99.7  77.2   0:36.78     0      4  11.9g  11.9g  11.9g      0      0      0 test_mtxmult_mp
 262222 admin     20   0   23.8g  11.9g      0 R  99.7  77.2   0:36.70     0      4  11.9g  11.9g  11.9g      0      0      0 test_mtxmult_mp
 262224 admin     20   0   23.8g  11.9g      0 R  99.7  77.2   0:36.48     0      4  11.9g  11.9g  11.9g      0      0      0 test_mtxmult_mp
 262225 admin     20   0   23.8g  11.9g      0 R  99.7  77.2   0:36.35     0      4  11.9g  11.9g  11.9g      0      0      0 test_mtxmult_mp
 262226 admin     20   0   23.8g  11.9g      0 R  99.7  77.2   0:36.25     0      4  11.9g  11.9g  11.9g      0      0      0 test_mtxmult_mp
 262223 admin     20   0   23.8g  11.9g      0 R  99.0  77.2   0:36.46     0      4  11.9g  11.9g  11.9g      0      0      0 test_mtxmult_mp
```

Incidentally, this shows that it's not meaningfull to add the RES value of different processes as that does not amount to the actual, physical combined memory used by all of them.

Let's repeat this case but allocating the initial matrix via mmap call.

```
top - 19:10:35 up 35 days, 20:24,  3 users,  load average: 7.56, 5.63, 11.13
Tasks: 223 total,  10 running, 213 sleeping,   0 stopped,   0 zombie
%Cpu(s): 99.6 us,  0.1 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.2 hi,  0.1 si,  0.0 st
MiB Mem :  15808.6 total,    295.9 free,   1157.5 used,  14355.2 buff/cache
MiB Swap:      0.0 total,      0.0 free,      0.0 used.   1685.7 avail Mem

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+  SWAP   CODE   DATA   USED   RSan   RSfd   RSlk   RSsh COMMAND
 262333 admin     20   0   23.8g  11.9g  11.9g R  99.5  77.2   3:08.98     0      4    328  11.9g     92   1816      0  11.9g test_mtxmult_mp
 262334 admin     20   0   23.8g  11.9g  11.9g R  99.5  77.2   2:49.54     0      4    328  11.9g     92    132      0  11.9g test_mtxmult_mp
 262337 admin     20   0   23.8g  11.9g  11.9g R  99.5  77.2   2:49.65     0      4    328  11.9g     92    132      0  11.9g test_mtxmult_mp
 262339 admin     20   0   23.8g  11.9g  11.9g R  99.5  77.2   2:49.66     0      4    328  11.9g     92    132      0  11.9g test_mtxmult_mp
 262340 admin     20   0   23.8g  11.9g  11.9g R  99.5  77.2   2:49.62     0      4    328  11.9g     92    132      0  11.9g test_mtxmult_mp
 262336 admin     20   0   23.8g  11.9g  11.9g R  99.0  77.2   2:49.33     0      4    328  11.9g     92    132      0  11.9g test_mtxmult_mp
 262338 admin     20   0   23.8g  11.9g  11.9g R  99.0  77.2   2:49.30     0      4    328  11.9g     92    132      0  11.9g test_mtxmult_mp
 262335 admin     20   0   23.8g  11.9g  11.9g R  98.0  77.2   2:49.05     0      4    328  11.9g     92    132      0  11.9g test_mtxmult_mp
```

Again we see the shift from almost all the memory being private (above) to almost all being shared.

However, an analogous conclusion can be drawn: It's not meaningful to add the values of USED for different processes; neither is it to add the values of RSsh: **They do not account for independent chunks of physical memory; we are overcounting!**

Additionally, notice that in this case the physical used memory reported by top is just a mere 1158MiB, a bit over 1.GiB, that's about 180MiB over the initial baseline which in this case was 974MiB.

Can we account for that? I don't know how.

Finally, notice that the used buff/cache mem has increased in this case by roughly 12000MiB, again of the order of the size of the initial matrix.

Conclusions^

This post provides a first hands-on introduction to concurrent programming. You can use it for small projects on most OSs like OSX, Linux, FreeBSD -notably not Windows, which has no fork capabilties.

As an introduction it definitely doesn't cover all there is about a practical guideline to the topic.

For instance, passing information between parent and children could be done through pipes (very limited). We used a more sensible approach through memoring sharing (shm, mmap,etc.).

A better alternative is to fine tune the creationg of the children through the clone family of systems calls. These allow for selectively sharing things like the heap or stack.

When sharing memory among the parent and children, with read and write privileges for all, an important complication arises: race condtions. This means, as a programmer, must make sure that no two processes attempt to modify the same memory location at the same time, or forget to unlock on time a descriptor thus blocking a second processes from using it, etc. Semaphores is a keyword for digging into this.

This is their typical day-to-day life when people say they are using pthreads! ;-)

A high-performance, highly portable and scalable way of passing information among children and with the parent is through the use of an API initially developed in the academy and that became a standard in some scenarios: the MPI (Message Passing Interface) library. This relies on processes. There are ports to MPI in many languages (C, C++, Fortran, Java,...).

Coming back to our example, just with fork and memory sharing, we could have allocated the initial matrix via mmap (see the commented out lines in the code) and make it read-only right after populating its values. After all, this is a chunk of memory that only requires reading, hence COW won't lead to duplicate it. That could be a way for reducing the memory footprint of the children and, in turn, that the whole program.

Waiting for our child processes might eventually also be done using the system calls select (*NIXs) or epoll (Linux), like when using sockets in network programming.

Furthermore, the design of the webserver nginx provides a somewhat different alternative to spawning and controlling children through async events.

I may apend an addendum on some usage details of the memory sharing library another day. The example of matrix multiplication together with the man pages should help you get started.

Hope this helped you start writing some concurrent programs and put to work all your multitasking capabilities.

Besides books, man pages and wikipedia can be a useful resource. Don't waste it. ;-)

References^

Here some additional, useful references.

1. https://techtalk.intersec.com/2013/07/memory-part-2-understanding-process-memory/
2. https://stackoverflow.com/questions/131303/how-can-i-measure-the-actual-memory-usage-of-an-application-or-process
3. https://stackoverflow.com/questions/118307/a-way-to-determine-a-processs-real-memory-usage-i-e-private-dirty-rss
4. https://stackoverflow.com/questions/64546808/hands-on-example-of-forks-copy-on-write-in-action?noredirect=1#comment114135505_64546808