PARALLEL // PROGRAMMING

Gokhan Unel / UC Irvine

ISOTDAQ 2013

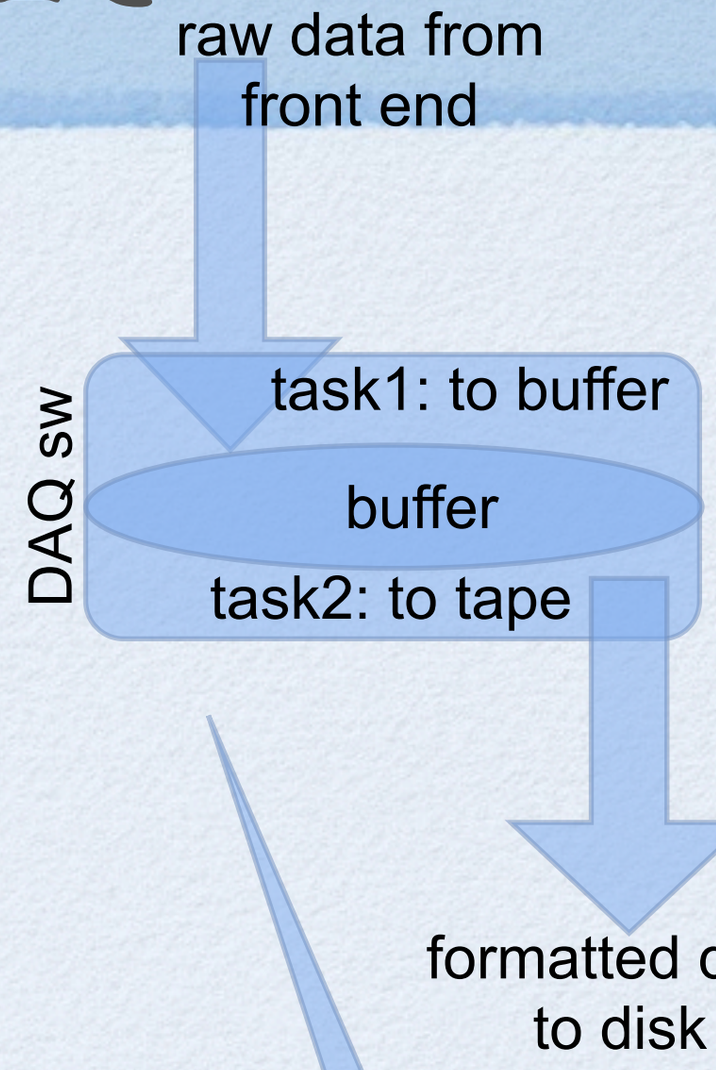
Thessaloniki, Greece

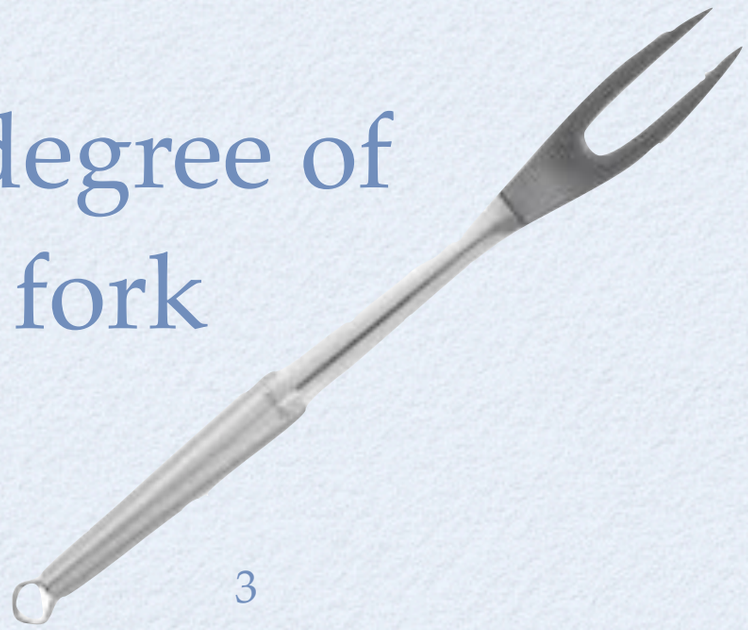
on //ism

- parallelism, parallel programming
  - to perform a number of tasks simultaneously
    - these can be calculations, or data transfer or…
- In Unix, many daemons run in parallel handling various tasks
  - Early hackers liked it! See XEROX CPV story in the bonus section

NA59 EXAMPLE

- Na59 DAQ software had 2 tasks:
  - task1: receive data, check its consistency put in a buffer.
  - task2: read data from buffer, record it on disk.
- We achieved this degree of parallelism with a fork
  - fork()





Note that task1 does port IO and a bit of calculation; task2 is mostly disk IO. They do not compete for the exact same resources

Who is who?

- Each process has a process ID #.
  - Say we have a program with PID i.
- After a fork() call,
  - there will be a second process with another PID j which we call the child.
  - The first process, will also continue to exist and to execute commands, we call it the mother.

WHY NOT TO FORK

- fork() system call is the easiest way of branching out to do 2 tasks concurrently. BUT
  - it creates a separate address space such that the child process has an exact copy of all the memory segments of the parent process.
  - This is “heavy” in terms of memory footprint
  - Forking is a “slow” operation.

The Lightweight Way

- Threads
  - smallest “executable” task
  - small footprint, quick to launch
  - contrary to processes, threads share all resources
    - memory (program code and global data)
    - open file/socket descriptors
    - signal handlers and signal dispositions
    - working environment (current directory, user ID, etc.)
  - included in posix standards
- Basic functions: start, stop, wait,...

WHY SHOULD I LEARN?

- CPU frequency scaling seems to have been saturated around 3GHz.
- We now see an increase of #cores / cpu
  - We need to learn // programming to get the maximum performance from any hardware
  - //programming also makes our sw more efficient.
- We want to be able to share LOTS of data and perform complicated tasks
  - real life examples from HEP experiments DAQ

STARTING A THREAD
pthread_create

- you have the man pages to remember the usage
  - man pthread_create

```
#include <pthread.h>

int  pthread_create(pthread_t *restrict thread,
                    const pthread_attr_t *restrict attr,
                    void *(*start_routine)(void *),
                    void *restrict arg);
```

- good luck with the short description...
  - learn (suffer) once, put in a wrapper library, use your library afterwards.
  - we will develop such a library during this lecture.

A WORKING EXAMPLE .

```
#include <errno.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <unistd.h>             /* POSIX et al */
#include <fcntl.h>
#include <signal.h>
#include <pthread.h>

#define TH_FAIL 1
#define TH_OK   0

void test_thread(char *arg)
{
     fprintf(stdout, "test_thread: errno=%d\n", errno);
     while (1) {
          sleep(1);
          fprintf(stdout, "%s\n", arg);
     }
}
```

includes

simple definitions

simple function to print the argument

```
int main(void)
{
     int prio;
     int status;

     char *data;
     long th_handle1, th_handle2, th_handle3;

     data = "GREEN";
     status = launch_thread(&th_handle1, test_thread, data);

     data = "orange";
     status = launch_thread(&th_handle2, test_thread, data);

    sleep(10);
    printf("test DONE !\n");

     exit(0);
}
```

- Our wrapper function: `launch_thread`
  - send : function to execute, argument to the function
  - receive : status, tid

A WORKING EXAMPLE ..

```
int launch_thread(long *th_handle,
                  void *start_func, void *param)
{
     pthread_t tid;
     pthread_attr_t attr;

#ifdef DEBUG
     if (param != NULL) {
          printf("sys_open_thread: arg is %d \n", *((int *) param));
     }
#endif

     pthread_attr_init(&attr);
     if (pthread_create(&tid, &attr, start_func, param) == -1) {
          printf("sys_open_thread: pthread_create error.\n");
          return (TH_FAIL);
     }
     *th_handle = (long int)tid;
     return (TH_OK);
}
```

ins & outs

create thread

return TID

- compile and link with -lpthread

```
gcc ex1.c -o ex1b -lpthread
```

demo 1

TAKE AWAY FROM EX1

- How to run multiple tasks in parallel.
- How to hide complex function calls in to a simple wrapper library.
- How to compile and execute a threaded program in Unix.

GET THE ID

- Each thread has a unique ID#: TID
  - a thread's TID is returned at the creation.
  - a thread's own TID is obtained with `pthread_self()`

- TIDs are of pthread_t which can be cast to a long unsigned int. In a thread, one can do
  - `printf("This is TID: %lu:\n",(unsigned long)pthread_self());`
    - could be useful in debugging...

TOOLS

- ps: process list, exists in all Unix variants
  - options to list processes including thread IDs, check the man page for your *nix.
    - linux, try: ps -eLf
- pstree: linux specific
- top: sort processes, exists in all Unix variants
  - Linux: H to turn on/off display of Threads

TID

```
UID        PID  PPID   LWP  C NLWP
fizikci    549   541   549  0    1
fizikci    560     1   560  0    1
fizikci    561     1   561  0    1
fizikci    568     1   568  0    1
fizikci    573     1   573  0    1
fizikci    583   549   583  0    2
fizikci    583   549   588  0    2
fizikci    587   583   587  0    1
```

```
[fizikci@hpfbu ~]$ pstree
init─┬─Terminal─┬─bash
     │          ├─bash───pstree
     │          ├─gnome-pty-helpe
     │          └─{Terminal}
     ├─3*[VBoxClient───{VBoxClient}]
     ├─VBoxClient
     ├─6*[agetty]
     ├─crond
     ├─dbus-daemon
     ├─dbus-launch
     ├─dhcpcd
     ├─gpg-agent
     ├─gvfs-fuse-daemo───3*[{gvfs-fuse-daemo}]
     ├─gvfsd
     ├─gvfsd-trash
     ├─slim─┬─X───2*[{X}]
     │      └─sh───xfce4-session─┬─Thunar
     │                           ├─xfce4-panel─┬─panel-6-systray
     │                           │             └─{xfce4-panel}
     │                           ├─xfdesktop───{xfdesktop}
     │                           ├─xfwm4
     │                           └─{xfce4-session}
     ├─sshd
     ├─syslog-ng───syslog-ng
     ├─udevd───2*[udevd]
     ├─xfce4-settings-
     ├─xfconfd
     └─xfsettingsd
```

TERMINATE A THREAD

pthread_cancel

- if the executed function exits or finishes, the thread exits as well.
- pthread_exit terminates the current thread.
- pthread_cancel nicely terminates any thread, it requires a TID to work on.
  - useful functions, we better add to our library

```
int exit_thread(long my_th_handler)
{
     if (my_th_handler == 0) {
          pthread_exit(0);      /* If self, just exit */
     }
     if (pthread_cancel((pthread_t)my_th_handler) != 0) {
          printf("exit_thread: cancel error\n");
          return (TH_FAIL);
     }
#ifdef DEBUG
     printf("killed! %d \n", (my_th_handler));
#endif
     return (TH_OK);
}
```

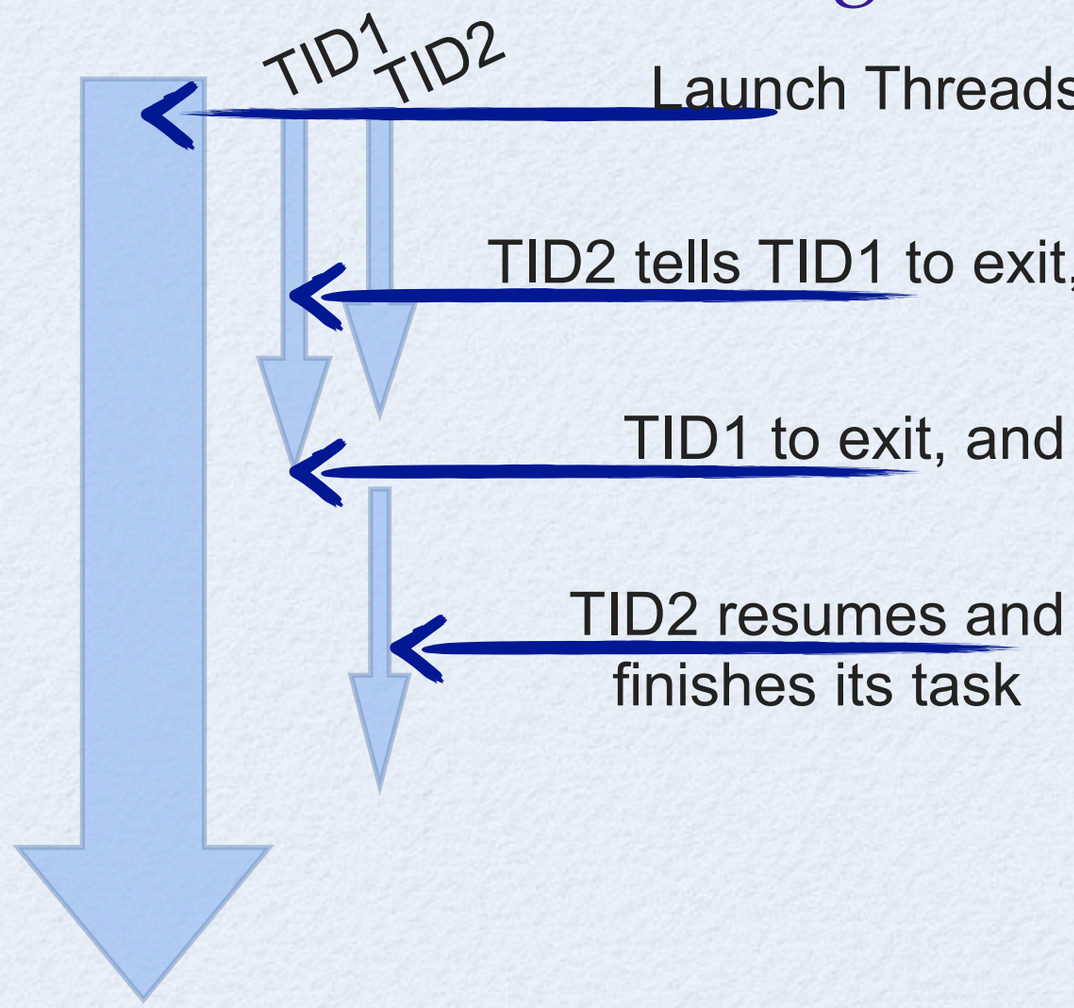
TAKE AWAY FROM EX2A

- Don't assume that you really killed a thread even if the system call returns success.
  - it will terminate the function whenever it is convenient.
- We have very fast computers, few microseconds means long time.
  - you will not have exact control if you do not wait for the actual termination.

WAIT FOR EXIT: JOIN

pthread_join

- The pthread_join() function suspends execution of the calling thread until the target thread exits,
  - unless the target thread has already terminated.




```
int waitfor_thread(long my_th_handler)
{
     int i, status;

     if (pthread_join((pthread_t)my_th_handler, (void **) &status) != 0) {
          printf("sys_waitfor_thread: join error\n");
          return (TH_FAIL);
     }
     return (TH_OK);

}
```

TO WAIT OR NOT ?

addition to main

```
dummy=0;

while (dummy<1E2) {
 if ( dummy==0) {
   printf ("Killing a TH.\n");
   status = exit_thread(th_handle2);
   printf ("Sent kill signal.\n");
   status = waitfor_thread(th_handle2);
   printf ("Killed.\n");
 }
 printf (" main active\n");
 usleep(10);
 dummy++;
}
```

- If we don't use the wait function, the "dead" thread will be still active for a while:
  - If not used, see that "orange" printed after "Killed."
  - If used, there is no such printout.

demo 2b

```
GREEN
orange
orange
Killing a TH.
Sent kill signal.
orange
GREEN
Killed.
 main active
GREEN
 main active
GREEN
```

```
Killing a TH.
Sent kill signal.
Killed.
 main active
GREEN
GREEN
 main active
orange
GREEN
GREEN
GREEN
 main active
GREEN
```

TAKE AWAY FROM EX2B

	+	-	Real life example
with wait function	Precision in task executions	•Becomes priority vs all other commands •Time consuming	ATM machine controls
without wait function	Allows other functions to continue in parallel	No accurate timing in execution	User interaction (e.g. display help)

IF TITANIC HAD AN AUTOPILOT

- Say we had a multi-threaded program responsible from piloting the ship
  - thread 1: weather conditions
  - thread 2: engine status
  - thread 3: helm control
  - thread 4: external sensors, cameras etc…

- tid4 notices the iceberg
- how to inform the rest of the working tasks ?
  - all threads should stop what they do, and act according to the new information.

Let them know: signal

pthread_kill

- Life is not synchronous ! But Asynchronous.
  - Things can happen at any time, a good software has to take into account such events.

- In computing, we use signals to mark such events.
  - for example, we could pause and continue a thread at our will.

PS: Don't let _kill fool you, it simply means send a signal.

```
int suspend_thread(long my_th_handler)
{
     if (pthread_kill(my_th_handler, SUSPEND_SIGNAL) == -1) {
          printf("suspend_thread: kill error\n");
          return (TH_FAIL);
     }
     return (TH_OK);
}
```

HANDLING THE SIGNALS

- Our library should be able to
  - receive a signal
  - do something about it

```
int thread_signal_handler(void)
{
     struct sigaction th_act;
     th_act.sa_handler = thread_signal_action;
     th_act.sa_flags= 0;
     if (sigaction(SUSPEND_SIGNAL, &th_act, (struct sigaction *) NULL)) {
          printf("sigaction: cant catch SUSPEND_SIGNAL\n");
          return (TH_FAIL);
     }
     pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);
     pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS, NULL);

     return TH_OK;
}
```

```
static void thread_signal_action(int signo)
{
     printf ("We got signal: %i\n",signo);
     if (signo == SUSPEND_SIGNAL) { sleep (1000000);}
     return;
}
```

Poor man's suspend.
We will revisit this.

```
void test_thread(char *arg)
{
     thread_signal_handler();
     fprintf(stdout, "test_thread: errno=%d\n", errno);
     while (1) {
          usleep(10000);
          fprintf(stdout, "%s\n", arg);
     }
}
```

executed function should call the signal handler

TESTING

```
dummy=0;

printf ("suspending TH2...\n");
suspend_thread(th_handle2);

sleep(1);
printf("test DONE !\n");
exit(0);
```

- modify the main to add the new function

```
GREEN
orange
GREEN
orange
GREEN
orange
GREEN
orange
suspending TH2...
We got signal: 4
GREEN
GREEN
GREEN
GREEN
GREEN
GREEN
GREEN
GREEN
GREEN
GREEN
```

no more orange !

demo 3

TAKE AWAY FROM EX3

- Singals are very useful to work in async mode.
  - but we can send 1 signal to 1 TID at a time
  - there is a way for “broadcasting”, we’ll see it later.

ALWAYS USE PROTECTION

- *problem*: as the resources are shared, a variable can be modified by multiple threads

```
int aninteger=0;

void test_thread(char *arg)
{
     fprintf(stdout, "test_thread: errno=%d\n", errno);
     while (1) {
          usleep(1e4);
          fprintf(stdout, "%s and i=%i\n", arg, aninteger);
          if (strcmp(arg,"GREEN") ) {aninteger++;} else {aninteger--;}
     }
}
```

```
orange and i=-2
GREEN and i=-2
orange and i=-2
GREEN and i=-1
GREEN and i=-2
orange and i=-2
GREEN and i=-2
orange and i=-2
test DONE !
An integer=-2
```

run1

```
GREEN and i=0
orange and i=-1
GREEN and i=0
orange and i=-1
GREEN and i=0
orange and i=-1
test DONE !
An integer=0
```

run2

```
orange and i=1
GREEN and i=2
orange and i=1
GREEN and i=2
orange and i=1
GREEN and i=2
orange and i=1
test DONE !
An integer=2
```

run3

- *solution*: use MUTual EXclusion as protection.
  - a MUTEX is like a safe box

CREATE AND LOCK

- There are 2 calls to create a mutex
  - `pthread_mutexattr_init ( pthread_mutexattr_t *attr);`
  - `pthread_mutex_init (pthread_mutex_t *mutex, const pthread_mutexattr_t *attr)`
- There are 2 calls to lock /unlock a mutex
  - `pthread_mutex_lock ( pthread_mutex_t *mutex);  // blocking`
  - `pthread_mutex_unlock ( pthread_mutex_t *mutex);`

```
void test_thread(char *arg)
{
    while (1) {
        printf ("This is %s, WILL wait...\n",arg);
        if (pthread_mutex_lock(&sys_th_mut) != 0) {
                printf("sys_th_action 1: mutex_lock error\n");
        }
        if (strcmp(arg,"GREEN")==0 ) {
         aninteger++;
         usleep(1e6);
         printf ("This is %s, increasing\n",arg);
        } else { // this is what orange does
         printf ("This is %s, decreasing\n",arg);
         aninteger--;
         usleep(1); // orange will run much much faster...
        }
        printf ("This is %s, waiting OVER... and i=%i\n",arg, aninteger);
        if (pthread_mutex_unlock(&sys_th_mut) != 0) { // both will unlock
                  printf("sys_th_action 4: mutex_unlock");
        }
    }
}
```

locking call, if the mutex is free lock will be rapidly achieved, if not, the function call will wait until the mutex becomes available.

increase the common variable for GREEN and decrease for orange, except use different sleep values….

unlock the mutex, free it for the next usage.

OUTPUT

```
gokhans-macbook-pro:isotdaq2013 ngu$ ./ex5b
This is GREEN, WILL wait...
This is orange, WILL wait...
This is GREEN, increasing
This is GREEN, waiting OVER... and i=1
This is GREEN, WILL wait...
This is orange, decreasing
This is orange, waiting OVER... and i=0
This is orange, WILL wait...
This is GREEN, increasing
This is GREEN, waiting OVER... and i=1
This is GREEN, WILL wait...
This is orange, decreasing
This is orange, waiting OVER... and i=0
This is orange, WILL wait...
This is GREEN, increasing
This is GREEN, waiting OVER... and i=1
This is GREEN, WILL wait...
This is orange, decreasing
This is orange, waiting OVER... and i=0
This is orange, WILL wait...
test DONE !
An integer=1
```

demo 5b

- although orange is much much faster, we see same number of GREEN and orange calls.
  - This is nice, and expected: lock call waits for the MUTEX to be available.
- one should destroy a mutex when it is not needed
  - pthread_mutex_destroy(*mutex)

TAKE AWAY FROM EX5

- Use a MUTEX when you need
  - to limit access to a particular data / counter etc...
  - to make a thread wait something to happen without polling
- polling is bad,
  - waste of cpu cycles
    - if you sleep between polls no give cpu, you will loose reaction time

CONDITION VARIABLES .

- These provide another way for synchronization
  - mutexes control access to data
  - condition variables control actual value of data
- Why we need condition variables ?
  - these provide a blocking function, at low cpu usage
  - otherwise, one would have to poll continuously
- Note
  - a condition variable also needs a mutex

CONDITION VARIABLES ..

- two new variable types
  - `pthread_cond_t th_cond;`
  - `pthread_condattr_t th_cond_attr;`

condition variable and its attributes

- Five function calls
  - `pthread_condattr_init (*cond_attr);`
  - `pthread_cond_init(*cond, *cond_attr);`
  - `pthread_cond_broadcast (*cond);`
  - `pthread_cond_wait(*cond, *mutex);`
  - `pthread_cond_destroy(*cond);`

initialize the attributes

initialize the variable

publish the variable

blocking wait call

remove when done

LIBRARY IMPROVEMENTS

- Lets use the condition variables and blocking calls to implement a suspend-resume function
  - each thread has to have its own mutex & cond_var.

```
#define TH_FAIL 1
#define TH_OK   0
#define SUSPEND_SIGNAL  SIGILL
#define MAX_THREAD 10

pthread_mutex_t sys_th_mut[MAX_THREAD];
pthread_mutexattr_t sys_th_mut_attr;
pthread_cond_t sys_th_cond[MAX_THREAD];
pthread_condattr_t sys_th_cond_attr;
pthread_t thread_id[MAX_THREAD];

int th_global_num = -1;
```

we need to monitor the #of active threads in launch and exit

  - thread_launch: initialize cond_vars & mutexes
  - the signal action: incorporate cond_vars & mutexes
  - need resume_thread function
  - waitfor_thread: incorporate cond_vars & mutexes

```
int launch_thread(long *th_handle,
                  void *start_func, void *param)
{
     pthread_t tid;
     pthread_attr_t attr;
#ifdef DEBUG
     if (param != NULL) {
          printf("launch_thread: arg is %d \n", *((int *) param));
     }
#endif
     th_global_num++;
     if (th_global_num == MAX_THREAD) {
          printf("sys_open_thread: Maximum Thread Limit");
          return (TH_FAIL);
     }

     pthread_attr_init(&attr);
     if (pthread_create(&tid, &attr, start_func, param) == -1) {
          printf("open_thread: pthread_create error.\n");
          return (TH_FAIL);
     }
     *th_handle = (long int)tid;
     thread_id[th_global_num] = tid;
     if (pthread_mutexattr_init(&sys_th_mut_attr) == -1) {
          printf("sys_open_thread: mutexattr_init error");
          return (TH_FAIL);
     }
     if (pthread_mutex_init(&sys_th_mut[th_global_num], &sys_th_mut_attr) == -1) {
          printf("sys_open_thread: mutex_init error");
          return (TH_FAIL);
     }
     if (pthread_condattr_init(&sys_th_cond_attr) == -1) {
          printf("sys_open_thread: condattr_init error");
          return (TH_FAIL);
     }
     if (pthread_cond_init(&sys_th_cond[th_global_num], &sys_th_cond_attr) == -1) {
          printf("sys_open_thread: cond_init error");
          return (TH_FAIL);
     }
     return (TH_OK);
}
```

new: keep track of the # of active threads

new: init a mutex / thread

new: init a cond_var / thread

```
static void thread_signal_action(int signo)
{
     int i = 0, j;
     pthread_t this_one;
     {
          i = 1;
          this_one = pthread_self();

          for (j = 0; j <= MAX_THREAD; j++)
               if (thread_id[j] == this_one) {

                    while (i) {
                         if (pthread_mutex_lock(&sys_th_mut[j]) != 0) {
                              printf("thred_action 1: mutex_lock error\n");
                         }
#ifdef DEBUG
                         printf("waiting. . . .\n");
#endif
                         if (pthread_cond_wait(&sys_th_cond[j], &sys_th_mut[j]) == -1) {
                              printf("sys_th_action 2: cond_wait\n");
                              /* unlocks the mutex in case of error */
                              if (pthread_mutex_unlock(&sys_th_mut[j]) != 0) {
                                   printf("sys_th_action 3: mutex_unlock\n");
                              }
                         } else
                              i = 0;
#ifdef DEBUG
                         printf("phew... that was long!\n");
#endif
                    }
                    /* unlocks after the release */
                    if (pthread_mutex_unlock(&sys_th_mut[j]) != 0) {
                         printf("sys_th_action 4: mutex_unlock\n");
                    }
               }
     }
     return;
}
```

```
static void thread_signal_action(int signo)
{
     printf ("We got signal: %i\n",signo);
     if (signo == SUSPEND_SIGNAL) { sleep (1000000);}
     return;
}
```

changes

```
int waitfor_thread(long my_th_handler)
{
     int i, status;

     if (pthread_join((pthread_t)my_th_handler, (void **) &status) != 0) {
          printf("waitfor_thread: join error\n");
          return (TH_FAIL);
     }
     for (i = 0; i <= MAX_THREAD; i++)
          if (thread_id[i] == my_th_handler) {

               if (pthread_cond_destroy(&sys_th_cond[i]) != 0) {
                    printf("sys_waitfor_thread: cond_destroy error");
                    return (TH_FAIL);
               }
               if (pthread_mutex_destroy(&sys_th_mut[i]) != 0) {
                    printf("sys_waitfor_thread: mutex_destroy error");
                    return (TH_FAIL);
               }
     }
     return (TH_OK);
}
```

new: clean the cond_vars and mutexes

new: resume function

```
int resume_thread(long my_th_handler)
{
     int i, the_one = 0;
     for (i = 0; i <= MAX_THREAD; i++) {
          if (thread_id[i] == my_th_handler) the_one = i;
     }

     if (pthread_cond_broadcast(&sys_th_cond[the_one]) == -1) {
          printf("sys_resume_thread: cond_broadcast error");
          return (TH_FAIL);
     }
#ifdef DEBUG
     printf("the thread %d  is  back!!!!", the_one);
#endif

     return (TH_OK);
}
```

```
int exit_thread(long my_th_handler)
{
     if (my_th_handler == 0) {
          th_global_num--;
          pthread_exit(0);      /* If self, just exit */
     }
     if (pthread_cancel((pthread_t)my_th_handler) != 0) {
          printf("exit_thread: cancel error\n");
          return (TH_FAIL);
     }
#ifdef DEBUG
     printf("killed! %d \n", (my_th_handler));
#endif
     th_global_num--;
     return (TH_OK);
}
```

new: keep track of the # of active threads

```
     data = "GREEN";
     status = launch_thread(&th_handle1, test_thread, data);
     data = "orange";
     status = launch_thread(&th_handle2, test_thread, data);
     sleep(2);
     printf ("suspending TH2...\n");
     suspend_thread(th_handle2);
     sleep(2);
     printf ("resuming TH2...\n");
     resume_thread(th_handle2);
     sleep(1);
     printf("test DONE !\n");
     exit(0);
```

modify the testing part to suspend-resume

demo 6

OUTLOOK

- multi-processing vs multi-threading
  - amount of shared resources (& available memory) justifies launching multi threads.
- limitations of multi threading / processing
  - Amdahl's Law: the speedup of a program due to parallelization can be no larger than the inverse of the portion of the program that is immutably sequential.
    - For example, if 50% of your program is not parallelizable, then you can only expect a maximum speedup of 2x
- processor affinity
  - a program or a thread can be locked to a particular CPU (or core). This will override the OS's scheduling scheme. The calls are OS dependent. *Linux*: taskset & sched_setaffinity for processes and pthread_setaffinity_np & pthread_attr_setaffinity_np, for *BSD check your manual.
- vector processing features in modern CPUs
  - vector processing: single operation on multiple data OR multiple operation on multiple data. example: scale a 1x100 vector by π . Knowing your hardware would allow writing more efficient software.

HOMEWORK

- Google about the following high level tools for // programming
  - Boost
  - TBB
  - PROOF
- Make the last demo work on your computer w/ 3 colors.
  - make a real library with a .c and .h file + test suite + readme
  - challenge: could you make a resume_all_threads function?

- References
  - http://linux.about.com/library/cmd/blcmdl2_sched_setscheduler.htm
  - Programming with POSIX Threads: David R. Butenhof
  - https://computing.llnl.gov/tutorials/pthreads

BONUS

slides

ON SCHEDULING

- For different processes
  - Cooperative multitasking
    - Processes voluntarily cede time to one another
  - Preemptive multitasking - for realtime applications *(your smart phone!)*
    - Operating system guarantees that each process gets a “slice” of time for execution and handling of external events (I/O) such as interrupts
- For different threads
  - SCHED_FIFO - first in first out
    - runs until either it is blocked by an I/O request, it is preempted by a higher priority process, or it calls sched_yield.
  - SCHED_RR - round robin
    - same as above, except that each process is only allowed to run for a maximum time quantum.
  - SCHED_OTHER (linux, not in bsd)
    - Linux default, for processes without any realtime requirements.

FORK() EXAMPLE

```
#include <stdio.h>      // printf, stderr, fprintf
#include <sys/types.h> // pid_t
#include <unistd.h>    // _exit, fork
#include <stdlib.h>    // exit
#include <errno.h>     // errno

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

   if (pid == -1) {
      fprintf(stderr, "can't fork, error %d\n",
errno);
      exit(EXIT_FAILURE);
   }

   if (pid == 0) {
      /* Child process:
       * If fork() returns 0, it is the child
process.
       */
      int j;
      for (j = 0; j < 15; j++) {
         printf("child: %d\n", j);
         sleep(1);
      }
      _exit(0);  /* Note that we do not use
exit() */
   } // end of child
   else
   {

      /* If fork() returns a positive number, we
are in the parent process
       * (the fork return value is the PID of the
newly created child process)
       */
      int i;
      for (i = 0; i < 10; i++)
      {
         printf("parent: %d\n", i);
         sleep(1);
      }
      exit(0);
   }// end of parent

   return 0;
}
```

```
USER   PID  %CPU %MEM      VSZ    RSS   TT  STAT STARTED      TIME COMMAND
ngu  46379   0.0  0.0  2432744    204 s006  S+    3:35PM   0:00.00 ./a.out
ngu  46378   0.0  0.0  2432744    484 s006  S+    3:35PM   0:00.00 ./a.out
```

CIRCA 1974

- One fine day, the system operator on the main CP-V software development system in El Segundo was surprised by a number of unusual phenomena. These included the following:
  - Tape drives would rewind and dismount their tapes in the middle of a job.
  - Disk drives would seek back and forth so rapidly that they would attempt to walk across the floor (see walking drives).
  - The card-punch output device would occasionally start up of itself and punch a 'lace card' (card with all positions punched). These would usually jam in the punch.
  - The console would print snide and insulting messages from Robin Hood to Friar Tuck, or vice versa.
- Naturally, the operator called in the operating-system developers. They found the bandit ghost jobs running, and killed them... and were once again surprised. When Robin Hood was gunned, the following sequence of events took place:

```
!X id1

id1: Friar Tuck... I am under attack!  Pray save me!
id1: Off (aborted)

id2: Fear not, friend Robin!  I shall rout the Sheriff
     of Nottingham's men!

id1: Thank you, my good fellow!
```

  - Each ghost-job would detect the fact that the other had been killed, and would start a new copy of the recently slain program within a few milliseconds. The only way to kill both ghosts was to kill them simultaneously (very difficult) or to deliberately crash the system.