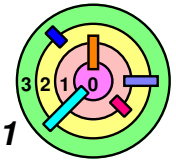
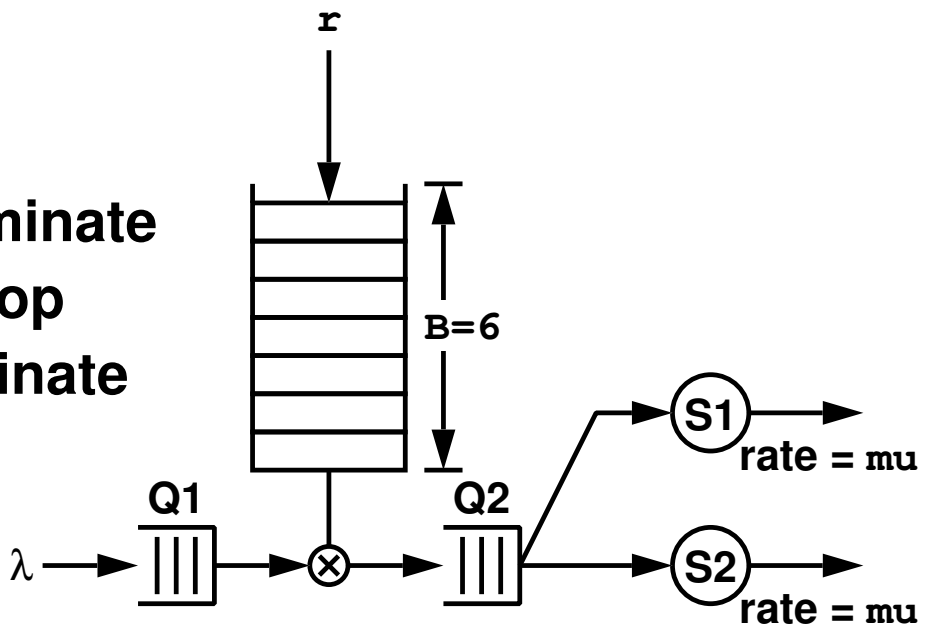
Warmup #2 (Part 2)

Bill Cheng

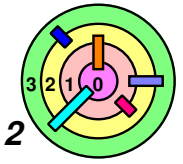
http://merlot.usc.edu/william/usc/



Handling <Ctrl+C> In Warmup #2

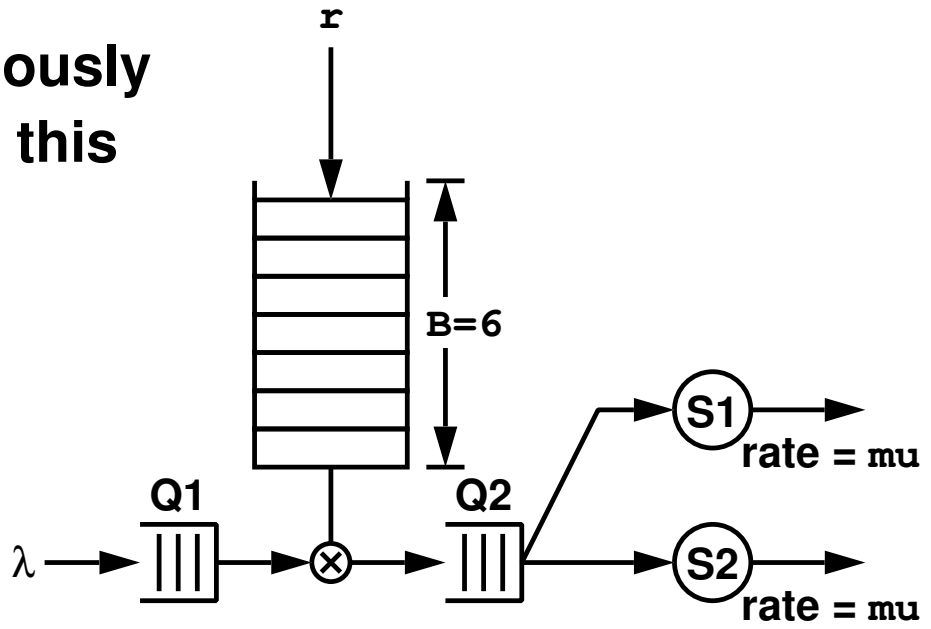


- <Ctrl+C>
  - packet arrival thread must stop generating packets and self-terminate
  - token depositing thread must stop generating tokens and self-terminate
  - a server thread must finish serving its current packet then self-terminate
  - no packets or tokens must arrive
  - need to take care of left-over packets that are stuck in Q1 and Q2
    - these are called *"removed packets"* because your simulator must remove them before you end your simulation
    - must distinguish 3 types of packets:
      1) *completed* packets
      2) *dropped* packets (packets that need > B tokens)
      3) *removed* packets

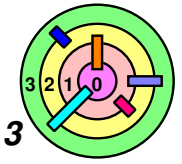


Handling <Ctrl+C> In Warmup #2

- You can catch <Ctrl+C> asynchronously
  - see lecture and understand why this is not a good idea
  - our recommendation is to *handle signals synchronously*
    - use a *signal-catching thread* and `sigwait()`
      - no signal handler at all



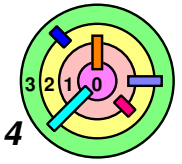
  - when <Ctrl+C> is caught, you should use the *pthread cancellation* mechanism to cancel packet and token threads
    - understanding thread cancellation will hopefully prevent you from make some mistakes in kernel assignment #1
      - although kernel cancellation is somewhat different from pthread cancellation



Designate A Thread To Catch A Signal

- Look at the man pages of `pthread_sigmask()` on Ubuntu 16.04
  - designate child thread to handler SIGINT
  - parent thread blocks SIGINT (some code deleted and use SIGINT instead of SIGQUIT):

```
#include <pthread.h>

int
main(int argc, char *argv[])
{
    pthread_t thread;
    sigset_t set;
    int s;

    sigemptyset(&set);
    sigaddset(&set, SIGINT);
    sigaddset(&set, SIGUSR1);
    s = pthread_sigmask(SIG_BLOCK, &set, NULL);
    if (s != 0)
        handle_error_en(s, "pthread_sigmask");

    s = pthread_create(&thread, NULL, &sig_thread, (void *) &set);
    if (s != 0)
        handle_error_en(s, "pthread_create");

    pause();
}
```



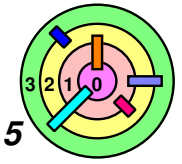
pthread_sigmask()

- Child thread example
  - child thread unblocks SIGINT

```
static void *
sig_thread(void *arg)
{
    sigset_t *set = arg;
    int s, sig;

    for (;;) {
        s = sigwait(set, &sig);
        if (s != 0)
            handle_error_en(s, "sigwait");
        printf("Signal handling thread got signal %d\n", sig);
    }
}
```

  - child thread is designated to handle SIGINT and SIGUSR1, no other thread will get SIGINT amd SIGUSR1

- Compile this code and run it
  - press <Ctrl+C> to see that this program cannot be killed with <Ctrl+C>
  - use "`kill -15`" command to kill this program

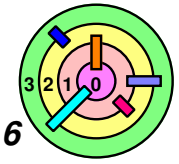


Example In Lecture

```
some_state_t state;
sigset_t set;

main() {
  pthread_t thread;
  sigemptyset(&set);
  sigaddset(&set,
            SIGINT);
  sigprocmask(
      SIG_BLOCK,
      &set, 0);
  // main thread
  //      blocks SIGINT
  pthread_create(
      &thread, 0,
      monitor, 0);
  long_running_proc();
}
```

```
void long_running_proc() {
  while (a_long_time) {
    pthread_mutex_lock(&m);
    update_state(&state);
    pthread_mutex_unlock(&m);
    compute_more();
  }
}

void *monitor() {
  int sig;
  while (1) {
    sigwait(&set, &sig);
    pthread_mutex_lock(&m);
    display(&state);
    pthread_mutex_unlock(&m);
  }
  return(0);
}
```

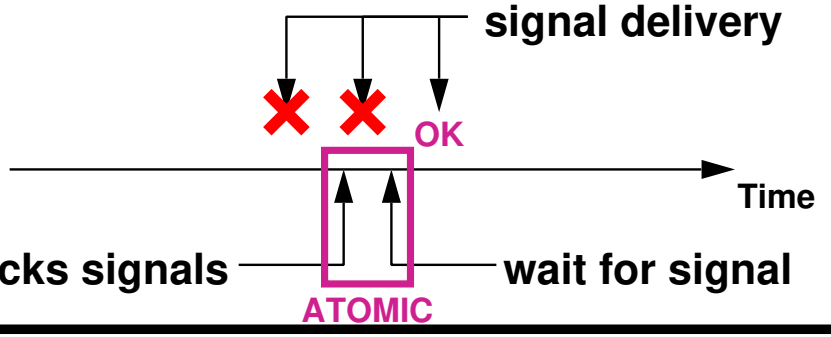
- this is the ***PREFERRED*** way to catch SIGINT for warmup2



sigwait()

```
int sigwait(sigset_t *set, int *sig)
```

- `sigwait()` blocks until a signal specified in `set` is received
  - return which signal caused it to return in `sig`
  - if you have a signal handler specified for `sig`, it will *not* get invoked when the signal is delivered
    - instead, `sigwait()` will return
- You should make sure that all the threads in your process have these signals blocked!
  - this way, when `sigwait()` is called, the calling thread temporarily becomes the *only* thread in the process who can receive the signal
- `sigwait(set)` *atomically unblocks* signals specified in `set` and *waits* for signal delivery



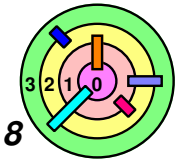
Warmup2 Catching <Ctrl+C> Made Easy

- Use a *<Ctrl+C>-catching thread* and *block SIGINT everywhere else*
  - when `sigwait()` returns because <Ctrl+C> has been pressed:

```
lock mutex
set global flag /* time to quit gracefully */
cancel packet and token threads
broadcast CV
unlock mutex
self terminate
```

    - according to spec, you must not cancel server threads

- Remember, if you don't use a <Ctrl+C>-catching thread, your regular threads can be borrowed to deliver signals
  - your code must deal with that and that can get very messy

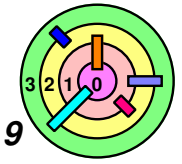


Warmup2 Cancellation Made Easy

- Only packet arrival and token depositing threads are allowed to be canceled
  - use a <Ctrl+C>-catching thread (i.e., use `sigwait()`)
  - handling clean up with `pthread_cleanup_push()` and `pthread_cleanup_pop()` can be messy
    - it's good to use them in general, but for warmup2, maybe we should try to make our lives easier by observing this:
    - first procedure of packet or token thread looks like:

```
do-forever
  sleep();
  lock mutex
  /* stuff */
  unlock mutex
end-do
```

    - if we have only one cancellation point, we know exactly where our threads will act on cancel

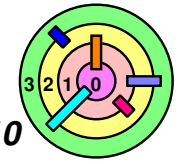


Warmup2 Cancellation Made Easy

- what if we change it to:

```
disable cancellation
do-forever
  enable cancellation
  sleep();
  disable cancellation
  lock mutex
  /* stuff */
  unlock mutex
end-do
```

- where can this thread act on cancel?
- would this make our thread *"unresponsive"* when <Ctrl+C> is pressed?
- there is a race condition in this code
  - what if the SIGINT-catching thread cancels this thread right when this thread is waiting for mutex lock?
  - must not create a new packet/token once you have printed the "SIGINT caught" message



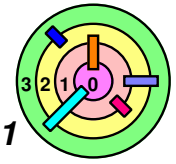
Statistics - Running Average

- Do not store all transmitted packets in a list and calculate statistics at the end of your simulation
  - it's a waste of memory
    - useful to learn how to write code with *small memory footprint*
  - keep a *running average* instead
    - just need one `int` and one `double`
      - `n` is the number of samples
      - `avg` is the average
      - when you get a new sample value, simply do:

```
avg = (n * avg + sample_value) / (n + 1)
n = n + 1
```

  - `avg` above is `E[X]`, how do you calculate $E[X^2]$?

```
avg2 = (n * avg2 + sample_value^2) / (n + 1)
n = n + 1
```

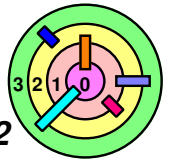


Statistics - Running Average

- can you use running averages for all these statistics?

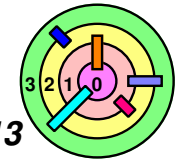
```
average packet inter-arrival time = <real-value>
average packet service time = <real-value>

average number of packets in Q1 = <real-value>
average number of packets in Q2 = <real-value>
average number of packets at S1 = <real-value>
average number of packets at S2 = <real-value>

average time a packet spent in system = <real-value>
standard deviation for time spent in system = <real-value>

token drop probability = <real-value>
packet drop probability = <real-value>
```

  - what should correspond to a sample value?
  - what types of packet must you consider?
    - avg packet inter-arrival time must consider ***all*** packets
    - other averages must only consider ***completed*** packets
    - packet drop probability is # dropped / # arrived



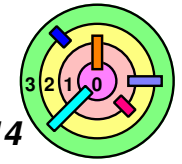
gdb

- When you get a breakpoint in `gdb`, what happened with other threads?
- If you break inside a signal handler and you type "`where`", how come `gdb` seems confused?



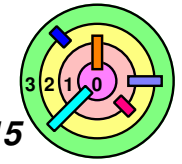
How To Learn New Concepts

- If there is a new concept that you are not familiar with, don't just try to write the final program
  - you won't know where the bugs are because you may not be clear about the concepts at multiple places
- Try writing small programs to test out ideas
  - try one idea at a time
  - use the debugger to get a better understanding of what's going on
  - then compile multiple ideas into one program and see if it works
- Ex:
  - `fork-wait.c`
  - `cat.c`
  - `redirect.c`
  - `thr-term.c`
  - `busywait.c, join.c`
  - `deadlock.c, trylock.c`
  - `whoopee.c`
  - `status-update.c`
  - `sigblock.c, sigwait.c`
  - `direct.c`
  - `cancel.c`
  - `three-threads.c`



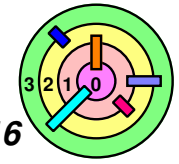
defs.h

```
#ifndef _DEFS_H_
#define _DEFS_H_

#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <signal.h>
#include <errno.h>

#include <pthread.h>

#ifndef NULL
#define NULL 0L
#endif /* ~NULL */

#endif /*_DEFS_H_*/
```



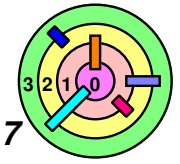
fork-wait.c

```
#include "defs.h"
#define NUM_CHILD 5
int sleep_time[NUM_CHILD], chd_num=0;
int main(int argc, char *argv[])
{
  srand48(time(0));
  for (chd_num=0; chd_num < NUM_CHILD; chd_num++) {
    sleep_time[chd_num] = lrand48() % 5000000;
    if (fork() == 0) {
      int pid=((int)getpid());
      printf("(Child) pid = %1d (0x%08x)\n", pid, pid);
      usleep(sleep_time[chd_num]);
      exit(child_pid+1);
    }
  }
  for (;;) {
    int pid=0, rc=0;
    if ((pid=wait(&rc)) == (-1)) break;
    printf("child %1d exited: 0x%08x.\n", pid, rc);
  }
  return 0;
}
```



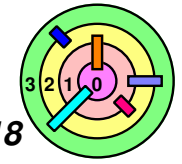
cat.c

```
#include "defs.h"
#define BUFSIZE 1024
int main(int argc, char *argv[])
{
  char buf[BUFSIZE];
  int n=0;
  const char *note="Write failed\n";

  while ((n = read(0, buf, sizeof(buf))) > 0)
    if (write(1, buf, n) != n) {
      (void)write(2, note, strlen(note));
      exit(1);
    }
  return 0;
}
```



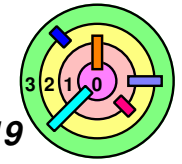
redirect.c

```
#include "defs.h"
int main(int argc, char *argv[])
{
  pid_t pid=(pid_t)0;
  if ((pid=fork()) == 0) {
    close(1);
    if (open("/tmp/Output",
        O_CREAT|O_WRONLY,
        0666) == -1) {
      perror("/tmp/Output");
      exit(1);
    }
    execl("/bin/date", "date", (char*)0);
    exit(1);
  }
  while(pid != wait(0)) ;
  return 0;
}
```



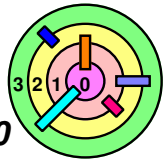
thr-term.c

```
#include "defs.h"
void *child(void *arg)
{
  if (*(int*)arg > 2) pthread_exit((void*)1);
  return((void*)2);
}
int main(int argc, char *argv[])
{
  pthread_t thread;
  void *result=NULL;
  pthread_create(&thread, 0, child, &argc);
  pthread_join(thread, (void**)&result);
  switch ((int)(long)result) {
  case 1: printf("result is 1\n"); break;
  case 2: printf("result is 2\n"); break;
  }
  return 0;
}
```



busywait.c

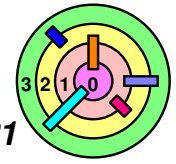
```
#include "defs.h"
#define NUM_THRS 100
int done[NUM_THRS];
pthread_t tid[NUM_THRS];
void *child(void *arg)
{
  int index=(int)(long)(arg);
  usleep(lrand48()%10000000);
  done[index] = 1;
  return 0;
}
int main(int argc, char *argv[])
{
  int i=0;
  memset(done, 0, sizeof(done));
  srand48(0);
  for (i=0; i < NUM_THRS; i++)
    pthread_create(&tid[i], 0, child, (void*)(long)i);
  waitall();
  return 0;
}
```



busywait.c

```
void waitall()
{
  for (;;) {
    int i=0, num_done=0;
    for (i=0; i < NUM_THRS; i++) {
      if (!done[i]) break;
      num_done++;
    }
    if (num_done == NUM_THRS) break;
  }
}
```

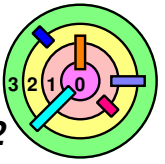
- Why is this busy wait?
  - the main thread is not doing anything useful
- Fix?
  - sleep for 100ms before checking
    - to avoid doing busy wait
    - not really a good solution



join.c

- Real fix
  - join with all threads

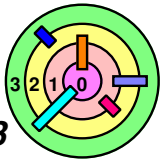
```
void waitall()
{
  int i=0;
  for (i=0; i < NUM_THRS; i++)
    pthread_join(tid[i], 0);
}
```



deadlock.c

Try to deadlock child thread and main thread

```
#include "defs.h"
void *child(void *arg)
{
  for (;;) { proc1(); }
  return((void*)0);
}

int main(int argc, char *argv[])
{
  pthread_t thread;
  srand48(time(0));
  pthread_create(&thread, 0, child, 0);
  for (;;) { proc2(); }
  return 0;
}
```

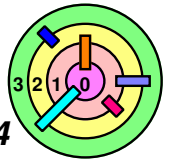


deadlock.c

```
pthread_mutex_t m1=PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t m2=PTHREAD_MUTEX_INITIALIZER;

void proc1() {
  pthread_mutex_lock(&m1);
  pthread_mutex_lock(&m2);
  printf("1");
  fflush(stdout);
  pthread_mutex_unlock(&m2);
  pthread_mutex_unlock(&m1);
  usleep(100000);
}
```

```
void proc2() {
  pthread_mutex_lock(&m2);
  pthread_mutex_lock(&m1);
  printf("2");
  fflush(stdout);
  pthread_mutex_unlock(&m1);
  pthread_mutex_unlock(&m2);
  usleep(100000);
}
```

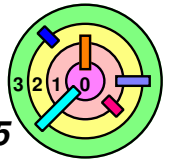
- Why no deadlock?
  - threads are alternating
- How to make it deadlock?
  - call `printf("-")` after locking `m1` in `proc1()` and call `printf("+")` after locking `m2` in `proc2()`
    - deadlock right away! (why?)



trylock.c

How to use `trylock()` to avoid deadlock

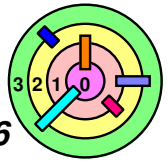
```
void proc2() {
  while (1) {
    pthread_mutex_lock(&m2);
    printf("+");
    fflush(stdout);
    if (!pthread_mutex_trylock(&m1))
      break;
    pthread_mutex_unlock(&m2);
  }
  printf("2");
  fflush(stdout);
  pthread_mutex_unlock(&m1);
  pthread_mutex_unlock(&m2);
  usleep(100000);
}
```



whoopee.c

- Let's catch SIGINT

```
#include "defs.h"
void handler(int signo)
{
  printf("Got signal %1d.  Whoopee!!\n", signo);
}
int main(int argc, char *argv[])
{
  sigset(SIGINT, handler);
  for (;;) { }
  return 1;
}
```



status-update.c

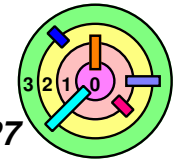
```
#include "defs.h"
typedef struct foo {
  int x, y;
} my_state;
my_state state;
int main() {
  state.x = state.y = 0;
  sigset(SIGINT, handler);
  long_running_proc();
  return 0;
}
```

```
void long_running_proc() {
  int i=0;
  for (i=0; i < 100; i++) {
    update_state(&state);
    compute_more();
  }
}
void handler(int signo) {
  display(&state);
}
```

```
void update_state(
    my_state *ptr) {
  ptr->x++;
  usleep(100000);
  ptr->y++;
}
```

```
void display(my_state *ptr) {
  printf("x = %1d\n", ptr->x);
  usleep(1000);
  printf("y = %1d\n", ptr->y);
}
```

```
void compute_more() { usleep(100000); }
```

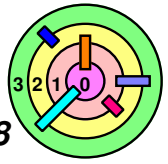


sigblock.c

- Let's block SIGINT

```
#include "defs.h"
typedef struct foo {
  int x, y;
} my_state;
my_state state;
sigset_t set;
int main() {
  state.x = state.y = 0;
  sigemptyset(&set);
  sigaddset(&set, SIGINT);
  sigset(SIGINT, handler);
  long_running_proc();
  return 0;
}
```

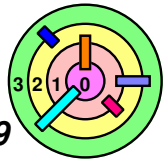
```
void long_running_proc() {
  int i=0;
  for (i=0; i < 100; i++) {
    sigset_t old_set;
    sigprocmask(SIG_BLOCK,
      &set, &old_set);
    update_state(&state);
    sigprocmask(SIG_SETMASK,
      &old_set, 0);
    compute_more();
  }
}
void handler(int signo) {
  display(&state);
}
```



sigwait.c

```
#include "defs.h"
typedef struct foo {
  int x, y;
} my_state;
my_state state;
sigset_t set;
pthread_mutex_t m=
PTHREAD_MUTEX_INITIALIZER;
int main() {
  pthread_t thr;
  state.x = state.y = 0;
  sigemptyset(&set);
  sigaddset(&set, SIGINT);
  sigprocmask(SIG_BLOCK,
    &set, 0);
  pthread_create(&thr, 0,
    monitor, 0);
  long_running_proc();
  return 0;
}
```

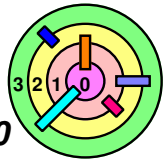
```
void long_running_proc() {
  int i=0;
  for (i=0; i < 100; i++) {
    pthread_mutex_lock(&m);
    update_state(&state);
    pthread_mutex_unlock(&m);
    compute_more();
  }
}
void *monitor(viod *arg) {
  int sig=0;
  for (;;) {
    sigwait(&set, &sig);
    pthread_mutex_lock(&m);
    display(&state);
    pthread_mutex_unlock(&m);
  }
  return 0;
}
```



direct.c

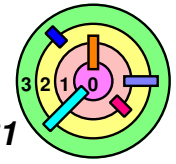
- Direct a thread to catch SIGINT
  - see "`man pthread_sigmask`" on Ubuntu 16.04

```
#include "defs.h"

#define handle_error_en(en, msg) \
  do { errno = en; perror(msg); exit(1); } while (0)

static void *
sig_thread(void *arg)
{
  sigset_t *set = arg;
  int s, sig;

  for (;;) {
    s = sigwait(set, &sig);
    if (s != 0)
      handle_error_en(s, "sigwait");
    printf("Signal handling thread got signal %d\n",
        sig);
  }
}
```



direct.c

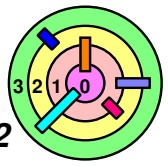
```
int
main(int argc, char *argv[])
{
  pthread_t thread;
  sigset_t set;
  int s;

  sigemptyset(&set);
  sigaddset(&set, SIGINT);
  sigaddset(&set, SIGUSR1);
  s = pthread_sigmask(SIG_BLOCK, &set, NULL);
  if (s != 0)
    handle_error_en(s, "pthread_sigmask");

  s = pthread_create(&thread, NULL, &sig_thread,
      (void *) &set);
  if (s != 0)
    handle_error_en(s, "pthread_create");

  pause();
}
```



cancel.c

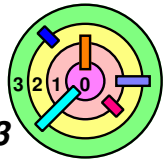
Cancellation

```
#include "defs.h"
#define NUM_THRS 10
pthread_t tid[NUM_THRS];
sigset_t set;
void cleanup(void *arg)
{
  int index=(int)(long)(arg);
  printf("Clean up thread %1d\n", index);
}
void *child(void *arg)
{
  pthread_cleanup_push(cleanup, arg);
  for (;;) {
    usleep(lrand48()%1000000);
  }
  pthread_cleanup_pop(0);
  return 0;
}
```



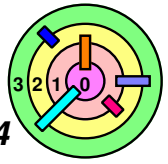
cancel.c

```
void waitall()
{
  int i=0;
  for (i=0; i < NUM_THRS; i++)
    pthread_join(tid[i], 0);
}
void *monitor(void *arg) {
  int i=0, sig=0;
  printf("Press <Ctrl+C>: ");
  fflush(stdout);
  sigwait(&set, &sig);
  printf("\nGot signal %1d\n", sig);
  for (i=0; i < NUM_THRS; i++) {
    while (tid[i] == 0) {
      usleep(100000);
    }
    pthread_cancel(tid[i]);
  }
  return 0;
}
```



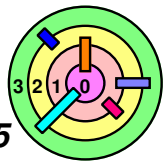
cancel.c

```
int main(int argc, char *argv[])
{
  pthread_t thr;
  int i=0;
  srand48(time(0));
  sigemptyset(&set);
  sigaddset(&set, SIGINT);
  sigprocmask(SIG_BLOCK, &set, 0);
  for (i=0; i < NUM_THRS; i++) {
    pthread_create(&tid[i], 0, child, (void*)(long)i);
  }
  pthread_create(&thr, 0, monitor, 0);
  waitall();
  pthread_join(thr, 0);
  return 0;
}
```



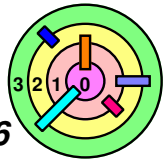
three-threads.c

```
/*
 * Demonstrate that only a thread with SIGINT unblocked
 *   will "see" SIGINT.
 * Currently, checking if num == 2 to unblock SIGINT
 *   for thread 2.  For all other threads, usleep() is
 *   never interrupted.
 *   Can change the comparison to 1 or 3 to demonstrate
 *     how to designate a different thread to only
 *     "see" SIGINT.
 */
#include <errno.h>
#include "defs.h"
sigset_t set;
void handler(int signo)
{
  printf("4");
  fflush(stdout);
}
```



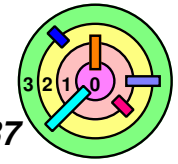
three-threads.c

```
void *child(void *arg) {
  int num=(int)(long)arg;
  if (num == 2) {
    sigprocmask(SIG_UNBLOCK, &set, 0);
  }
  for (;;) {
    printf("%1d", num);
    fflush(stdout);
    if (usleep(500000) != 0) {
      if (errno == EINTR) {
        printf("\nthread %1d interrupted...\n", num);
        fflush(stdout);
      }
    }
  }
  return 0;
}
```



three-threads.c

```
int main(int argc, char *argv[])
{
  int i=0;
  pthread_t t[3];
  sigset(SIGINT, handler);
  sigemptyset(&set);
  sigaddset(&set, SIGINT);
  sigprocmask(SIG_BLOCK, &set, 0);
  for (i=0; i < 3; i++)
    pthread_create(&t[i], 0, child, (void*)(long)(i+1));
  for (;;) {
    printf("0");
    fflush(stdout);
    usleep(500000);
    if (errno == EINTR) {
      printf("\nMain thread interrupted...\n");
      fflush(stdout);
    }
  }
  return 0;
}
```



Kernel Programming Assignments

Bill Cheng

http://merlot.usc.edu/william/usc/