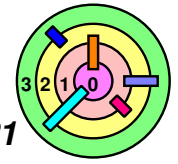
Signals and Blocking System Calls

- What if a signal is generated while a process is blocked in a system call?
  1) deal with it when the system call completes
  2) interrupt the system call, deal with signal, resume system call
  3) interrupt system call, deal with signal, return from system call with indication that something happened
- most systems choose (3)
  - `errno` sets to EINTR to mean that the system call was not completed because it was interrupted by a signal
    - this is the `errno` for the thread that was "deviated" to execute the signal handler
  - this may be the reason why `pthread_cond_wait()` may return "spontaneously" even when the CV has not been signaled/broadcasted
    - what if this thread was borrowed to deliver a signal?

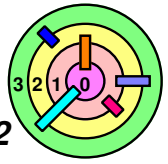


Copyright © William C. Cheng

Interrupted System Calls

```
while(read(fd, buffer, buf_size) == -1) {
  if (errno == EINTR) {
    /* interrupted system call; try again */
    continue;
  }
  /* the error is more serious */
  perror("big trouble");
  exit(1);
}
```

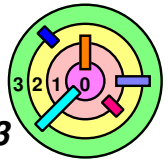
- need to check the return value of `read()` because `read()` can return when less than `buf_size` bytes have been read
- can use similar code for `write()`
  - same consideration as `read()`
- please note that the above code is incomplete, it needs to handle the case where `read()` return 0 to mean end-of-input



Copyright © William C. Cheng

Interrupted While Underway

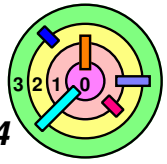
```
remaining = total_count; /* write this many bytes */
bptr = buf;              /* starting from here */
for ( ; ; ) {
  num_xfrd = write(fd, bptr, remaining);
  if (num_xfrd == -1) {
    if (errno == EINTR) {
      /* interrupted early */
      continue;
    }
    perror("big trouble");
    exit(1);
  }
  if (num_xfrd < remaining) {
    /* interrupted in the middle of write() */
    remaining -= num_xfrd;
    bptr += num_xfrd;
    continue;
  }
  /* success! */
  break;
}
```



Copyright © William C. Cheng

Interrupted System Calls

- If a thread can "see" signal delivery (i.e., "deviated to execute a signal handler"), every `read()` and `write()` call in that thread needs to look like the previous slides
  - much easier if you use a *signal-catching thread*
    - and *block all appropriate signals* in all "regular" threads
    - for warmup2, it is strongly encouraged that you do it this way to catch <Ctrl+C> and avoid using a signal handler
      - when `sigwait()` returns, lock mutex, set global flag, cancel packet arrival and token depositing threads, broadcast CV, unlock mutex, and self-terminate
      - according to spec, you must not cancel server threads
      - will talk about cancellation shortly



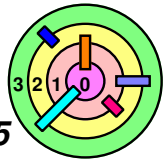
Copyright © William C. Cheng

Inside A Signal Handler

Which library routines are safe to use *within* signal handlers?

access	dup2	getgroups	rename	sigprocmask	time
aio_error	dup	getpgrp	rmdir	sigqueue	timer_getoverrun
aio_suspend	execle	getpid	sem_post	sigsuspend	timer_gettime
alarm	execve	getppid	setgid	sleep	timer_settime
cfgetispeed	_exit	getuid	setpgid	stat	times
cfgetospeed	fcntl	kill	setsid	sysconf	umask
cfsetispeed	fdatasync	link	setuid	tcdrain	uname
cfsetospeed	fork	lseek	sigaction	tcflow	unlink
chdir	fstat	mkdir	sigaddset	tcflush	utime
chmod	fsync	mkfifo	sigdelset	tcgetattr	wait
chown	getegid	open	sigemptyset	tcgetpgrp	waitpid
clock_gettime	geteuid	pathconf	sigfillset	tcsendbreak	write
close	getgid	pause	sigismember	tcsetattr	
creat	getoverrun	pipe	sigpending	tcsetpgrp	

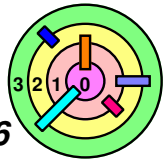
Note: in general, you should only do what's absolutely necessary inside a signal handler (and figure out where to do the rest)



Copyright © William C. Cheng

Cancellation

- The user pressed <Ctrl+C>
  - or a request is generated to terminate the process
  - the chores being performed by the remaining threads are no longer needed
  - in general, we may just want to cancel a bunch of threads and not the entire process
- Concerns
  - getting cancelled at an inopportune moment
    - a mutex left locked
    - a data structure is left in an inconsistent state
      - e.g., you get a cancellation request when you are in the middle of a `insert()` operation into a doubly-linked list and `insert()` is protected by a mutex
  - cleaning up (free up resources that only this thread can free up)
    - memory leaks
    - unlocking mutex if locked



Copyright © William C. Cheng

Cancellation State & Type

- Send cancellation request to a thread *(this is a non-blocking call)*

```
pthread_cancel(thread)
```

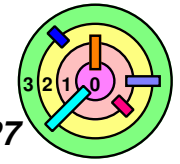
- Cancels enabled or disabled

```
int pthread_setcancelstate(
    { PTHREAD_CANCEL_DISABLE,
      PTHREAD_CANCEL_ENABLE},
    &oldstate)
```

- Asynchronous vs. deferred cancels

```
int pthread_setcanceltype(
  { PTHREAD_CANCEL_ASYNCHRONOUS,
    PTHREAD_CANCEL_DEFERRED},
  &oldtype)
```

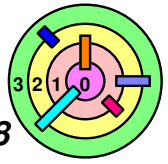
- By default, a thread has cancellation *enabled* and *deferred*
  - it's for a good reason
  - if you are going to change it, you must ask yourself, "Why?" and "Are you sure this is really a good idea?"



Copyright © William C. Cheng

POSIX Cancellation Rules

- **POSIX threads cancellation rules (part 1):**
  - when `pthread_cancel()` gets called, the target thread is marked as having a *pending cancel*
    - the thread that called `pthread_cancel()` does not wait for the cancel to take effect
  - if the target thread has cancellation *disabled*, the target thread stays in the pending cancel state
  - if the target thread has cancellation *enabled* ...
    - if the cancellation type is *asynchronous*, the target thread immediately *acts on cancel* (i.e., cancellation is "delivered" by "deviating" the thread to call `pthread_exit()`)
    - if the cancellation type is *deferred*, cancellation is *delayed* until it reaches a *cancellation point* in its execution
      - cancellation points correspond to points in the thread's execution at which it is safe to *act on cancel*

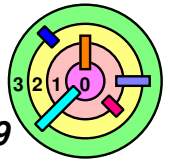


Copyright © William C. Cheng

Cancellation Points

```
aio_suspend
close
creat
fcntl (when F_SETLCKW
       is the command)
fsync
mq_receive
mq_send
msync
nanosleep
open
pause
pthread_cond_wait
pthread_cond_timedwait
pthread_join
pthread_testcancel
read
sem_wait
sigsuspend
sigtimedwait
sigwait
sigwaitinfo
sleep
system
tcdrain
wait
waitpid
write
```

- `pthread_mutex_lock()` is not on the list!
- `pthread_testcancel()` creates a cancellation point
  - useful if a thread contains no other cancellation point

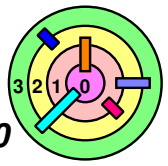


Copyright © William C. Cheng

POSIX Cancellation Rules

- POSIX threads cancellation rules (part 2):
  - when a thread *acts on cancel*
    - it calls `pthread_exit()`
    - in `pthread_exit()`, it first walks through a *stack* of *cleanup handlers*
      - when stack is empty, the thread goes into the zombie state
    - remember that the thread that called `pthread_cancel()` does not wait for the cancel to take effect
      - it may join and wait for the target thread to terminate

```
pthread_cleanup_push(
          (void)(*routine)(void *),
          void *arg)
pthread_cleanup_pop(int execute)
```



Copyright © William C. Cheng

Example

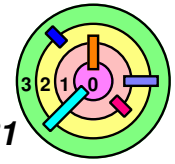
```
list_item_t list_head;

void *GatherData(void *arg) {
   list_item_t *item;
   item = (list_item_t*)malloc(sizeof(list_item_t));

   // GetDataItem() contains many cancellation points
   GetDataItem(&item->value);

   insert(item); // add item to a global list
   printf("Done.\n");
   return 0;
}
```

- How can this thread control when it acts on cancel?
  - so it doesn't leak memory



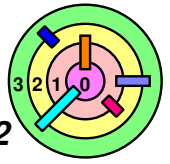
Copyright © William C. Cheng

Example

```
list_item_t list_head;

void *GatherData(void *arg) {
   list_item_t *item;
   item = (list_item_t*)malloc(sizeof(list_item_t));
   pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, 0);
   // GetDataItem() contains many cancellation points
   GetDataItem(&item->value);
   pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, 0);
   insert(item); // add item to a global list
   printf("Done.\n");
   return 0;
}
```

- How can this thread control when it acts on cancel?
  - so it doesn't leak memory
  - although this implementation is technically "correct", long delay may not be acceptable
    - need to *respond in a timely manner*



Copyright © William C. Cheng

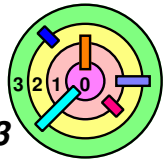
Example

```
list_item_t list_head;

void *GatherData(void *arg) {
   list_item_t *item;
   item = (list_item_t*)malloc(sizeof(list_item_t));
   pthread_cleanup_push(free, item);
   // GetDataItem() contains many cancellation points
   GetDataItem(&item->value);

   insert(item); // add item to a global list
   printf("Done.\n");
   return 0;
}
```

- Can act on cancel inside `GetDataItem()`
  - in this case, will invoke `free(item)`
  - in C library, `free()` is defined as: `void free(void *ptr);`
    - perfectly matches the argument types for `pthread_cleanup_push()`



Copyright © William C. Cheng

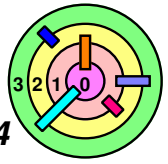
Example

```
list_item_t list_head;

void *GatherData(void *arg) {
   list_item_t *item;
   item = (list_item_t*)malloc(sizeof(list_item_t));
   pthread_cleanup_push(free, item);
   // GetDataItem() contains many cancellation points
   GetDataItem(&item->value);

   insert(item); // add item to a global list
   printf("Done.\n");
   return 0;
}
```

- What if it acts on cancel inside `printf()`
  - will end up calling `free(item)` twice
    - can cause segmentation fault later



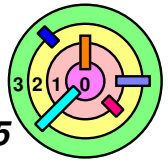
Copyright © William C. Cheng

Example

```
list_item_t list_head;

void *GatherData(void *arg) {
   list_item_t *item;
   item = (list_item_t*)malloc(sizeof(list_item_t));
   pthread_cleanup_push(free, item);
   // GetDataItem() contains many cancellation points
   GetDataItem(&item->value);
   pthread_cleanup_pop(0);
   insert(item); // add item to a global list
   printf("Done.\n");
   return 0;
}
```

- What if it acts on cancel inside `printf()`
  - will end up calling `free(item)` twice
    - can cause segmentation fault later
  - pop `free(item)` off the cleanup stack



Copyright © William C. Cheng

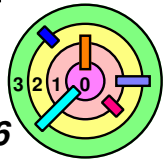
Example

```
list_item_t list_head;

void *GatherData(void *arg) {
   list_item_t *item;
   item = (list_item_t*)malloc(sizeof(list_item_t));
   pthread_cleanup_push(free, item); // {
   // GetDataItem() contains many cancellation points
   GetDataItem(&item->value);
   pthread_cleanup_pop(0);  // }
   insert(item); // add item to a global list
   printf("Done.\n");
   return 0;
}
```

must match up *(like a pair of brackets)*

- `pthread_cleanup_push()` and the corresponding `pthread_cleanup_pop()` must match up (like a pair of brackets)
  - must not call `pthread_cleanup_push()` in one function and call the corresponding `pthread_cleanup_pop()` in another
    - compile-time error



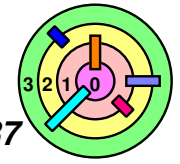
Copyright © William C. Cheng

Cancellation and Cleanup

```
void close_file(int fd) {
  close(fd);
}

fd = open(file, O_RDONLY);
pthread_cleanup_push(close_file, fd);
while(1) {
  read(fd, buffer, buf_size);
  // ...
}
pthread_cleanup_pop(0);
```

- should close any opened files when you clean up
- `int` is compatible with `void*`
  - well, sort of
  - `void*` can be a 64-bit quantity, so may need to be careful (best to be explicit)



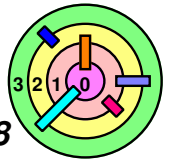
Copyright © William C. Cheng

Cancellation and Conditions

```
pthread_mutex_lock(&m);
pthread_cleanup_push(CleanupHandler, argument);

while(should_wait)
  pthread_cond_wait(&cv, &m);
// ... (code containing other cancellation points)
pthread_cleanup_pop(0);
pthread_mutex_unlock(&m);
```

- should `CleanupHandler()` call `pthread_mutex_unlock()`?
  - remember, if the thread is canceled between `push()` and `pop()`, we need to ensure that the mutex is *locked*
  - `pthread_cond_wait()` is a cancellation point
    - must not unlock the mutex twice!
- should `CleanupHandler()` call `pthread_mutex_lock()` then call `pthread_mutex_unlock()`?
  - what if the mutex is locked?
- application cannot solve this problem since there is *no way to check if a mutex is locked or not*



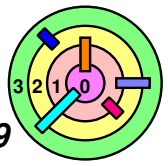
Copyright © William C. Cheng

Cancellation and Conditions

```
pthread_mutex_lock(&m);
pthread_cleanup_push(pthread_mutex_unlock, &m);

while(should_wait)
  pthread_cond_wait(&cv, &m);
// ... (code containing other cancellation points)
pthread_cleanup_pop(1);
```

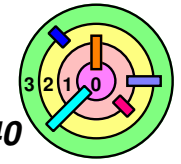
- pthreads library implementation ensures that a thread, when acting on a cancel inside `pthread_cond_wait()`, would first lock the mutex, before calling the cleanup routines
  - this way, the above code would work correctly



Copyright © William C. Cheng

Warmup2 Cancellation

- Only packet arrival and token depositing threads are allowed to be canceled
  - use a <Ctrl+C>-catching thread (i.e., use `sigwait()`)
  - make cancellation requests only when mutex is *locked*
    - this makes it impossible for those threads to act on cancel when they have the mutex locked
  - can make the following simplification for these two threads:
    - at the start of their first procedures, *disable cancellation*
    - right before calling `usleep()`, *enable cancellation*
    - when `usleep()` returns, *disable cancellation again*
    - this way, during the time the mutex is locked, cancellation is always disabled
      - therefore, don't have to worry about using cleanup routines to unlock mutex
    - but didn't we just say that it's not a good idea to disable cancellation?
    - also, need to take care of a *race condition*



Copyright © William C. Cheng

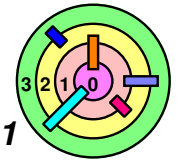
Cancellation & C++

```
void tcode() {
  A a1;
  pthread_cleanup_push(handler, 0);
  foo();
  pthread_cleanup_pop(0);
}

void foo() {
  A a2;
  pthread_testcancel();
}
```

- are the destructors of `a1` and `a2` getting called?
  - not sure
  - they should get called
  - some C++ implementation does not do this correctly!

⇨ Note: current C++ standard also does ***not*** support thread cancellation
- standard C++ threads must self-terminate!



Copyright © William C. Cheng