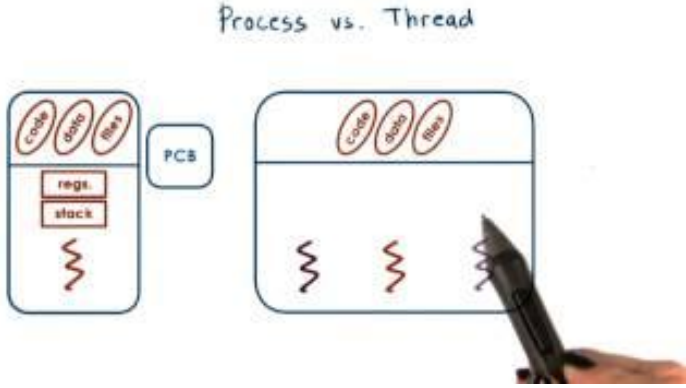


CSc 360
Operating Systems
Threads

Wenjun Yang

Fall 2025

Review: process

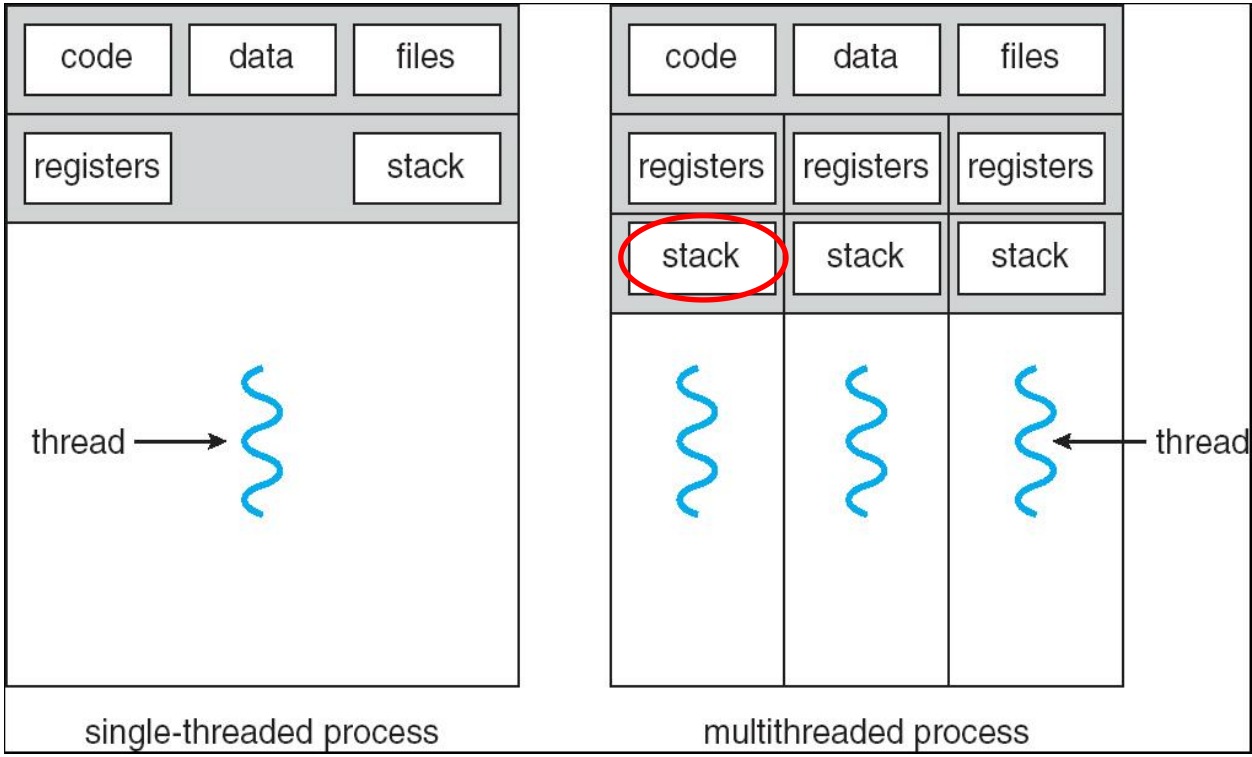
Assn 1 Due Tomorrow!
1st Midterm Coverage!

- Process
  - a running program + allocated resources
- Process scheduling
  - how to handle many processes (PCB)
- Process operation: play around in P1!
  - create processes and load a new program
- Process communication
  - shared memory vs message passing

Program, process, thread

- In one process
  - easy to share
- Btw processes
  - multitasking
- Best of both
  - thread
    - one process
    - multitasking

Recall stack?



Q: browsers to use multi-process?

Thread example 1

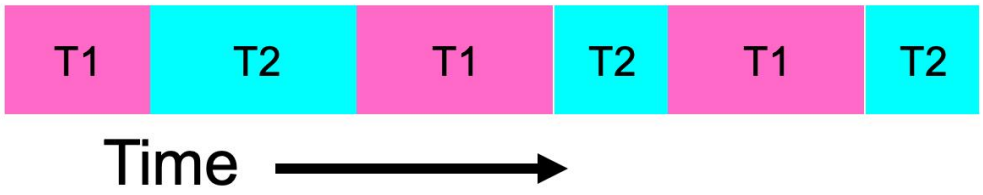
- Imagine the following program:
  - main() {
  - ComputePI(“pi.txt”);
  - PrintClassList(“classlist.txt”);
  - }

- What is the behavior here?

Thread example 2

- Version of program with threads (loose syntax):
  - main() {
  - create_thread(ComputePI, “pi.txt”);
  - create_thread(PrintClassList, “classlist.txt”);
  - }

- What is the behavior here?

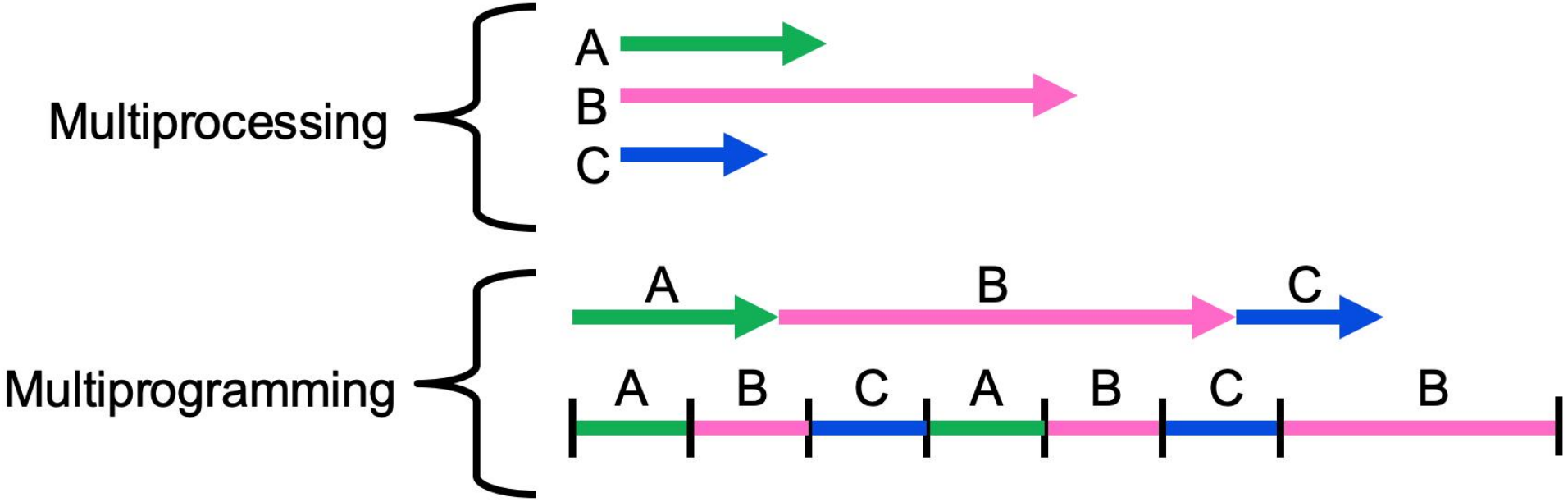


Threads

- Thread
  - a basic unit of CPU utilization
    - thread state, program counter, register set, stack
  - share with other threads in the **same** process
    - code, data, opened files, signals, etc
- Benefits
  - responsiveness: multithreading
  - resource sharing, efficiency, MC/MP platforms

Q: potential problems?

Multiprocessing, Multiprogramming, and Multithreading



Single-Threaded Program

Time

```
|   int x;
|   x = 20;
|   int y;
|   y = 50;
|   int sum;
|   sum = x + y;
```

Multi-Threaded Program

```
        |   int x;                  |  int a;
        |   x = 20;                 |  a = 3;
        |   int y;                  |  int b;
Time    |   y = 50;                 |  b = 5;
        |   int sum;                |  int product;
        |   sum = x + y;            |  product = a * b;
```

Parallel Execution

Multi-Threaded Program

```
Time
```

```
int x;

x = 20;
int y;

y = 50;

int sum;
sum = x + y;
```

```
int a;
a = 3;

int b;

b = 5;
int product;

product = a * b;
```

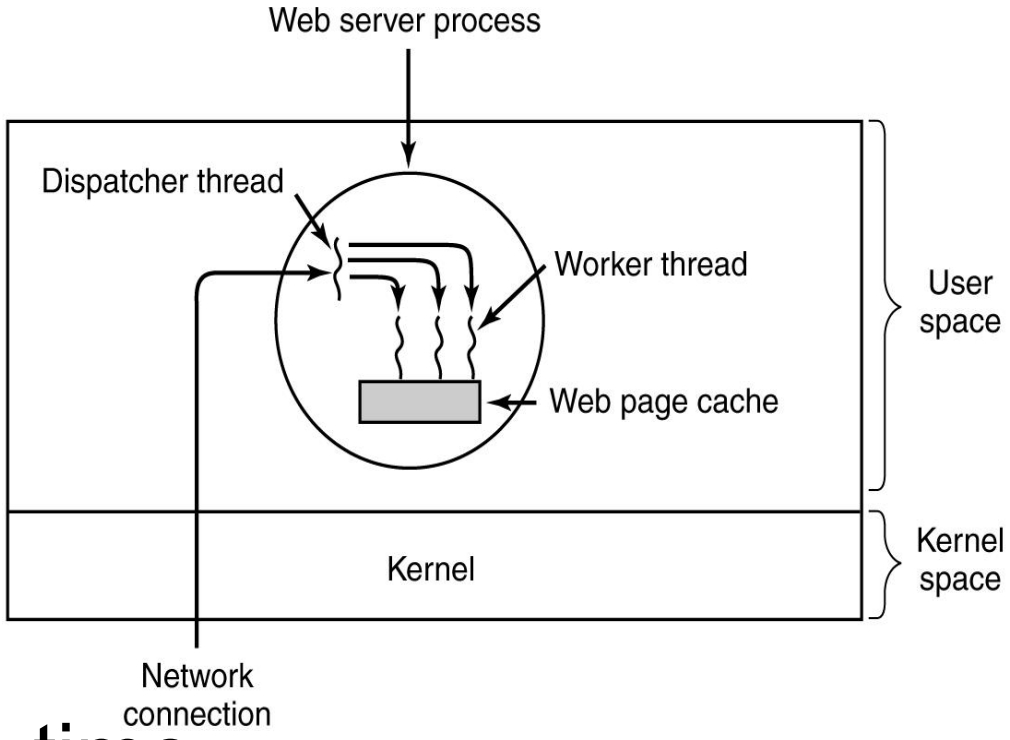
Concurrent But Not Parallel Execution

Single-threaded Web server

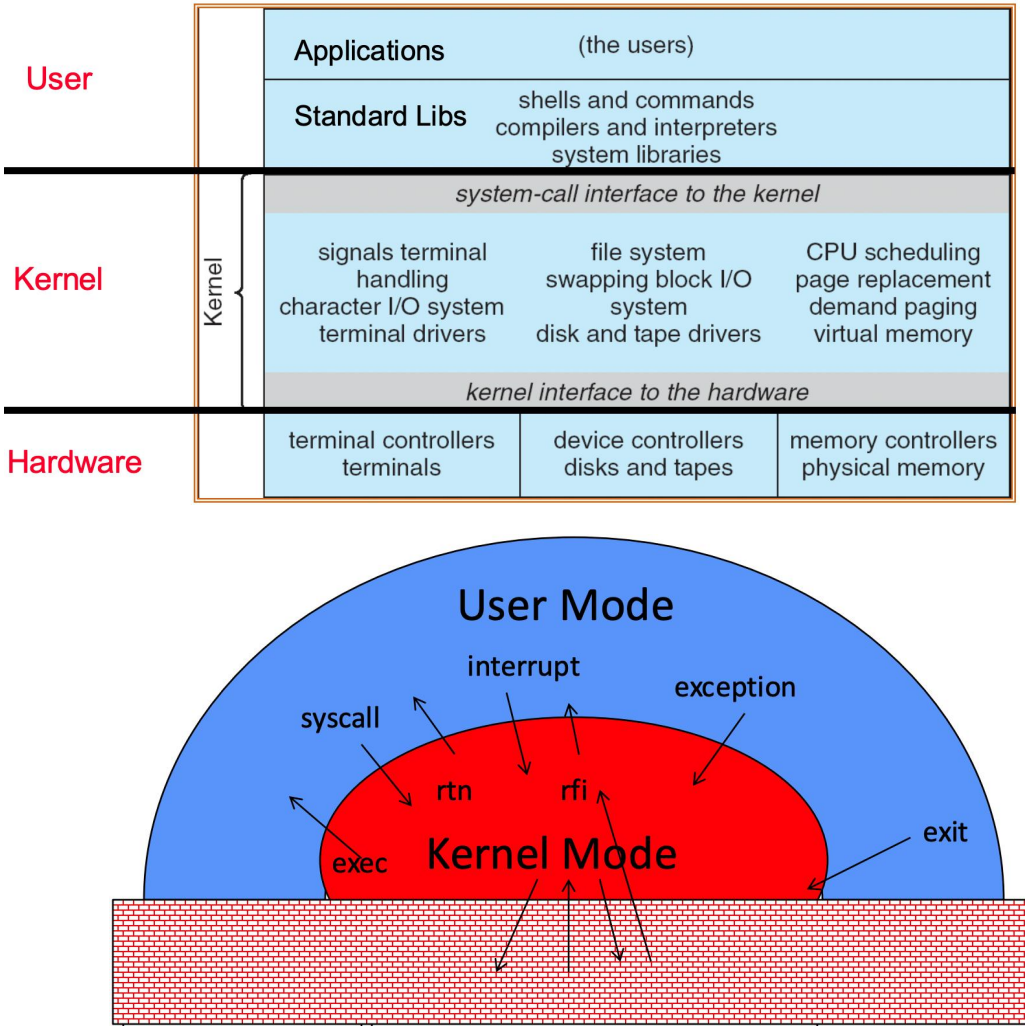
- Web server with cache and disk
  - wait for a request
  - process the request
    - check cache; if hit, break
    - otherwise, retrieve from disk (relatively slow)
  - respond the request
- One request at a time
  - or create a new process on each request
    - expensive!

Multi-threaded Web server

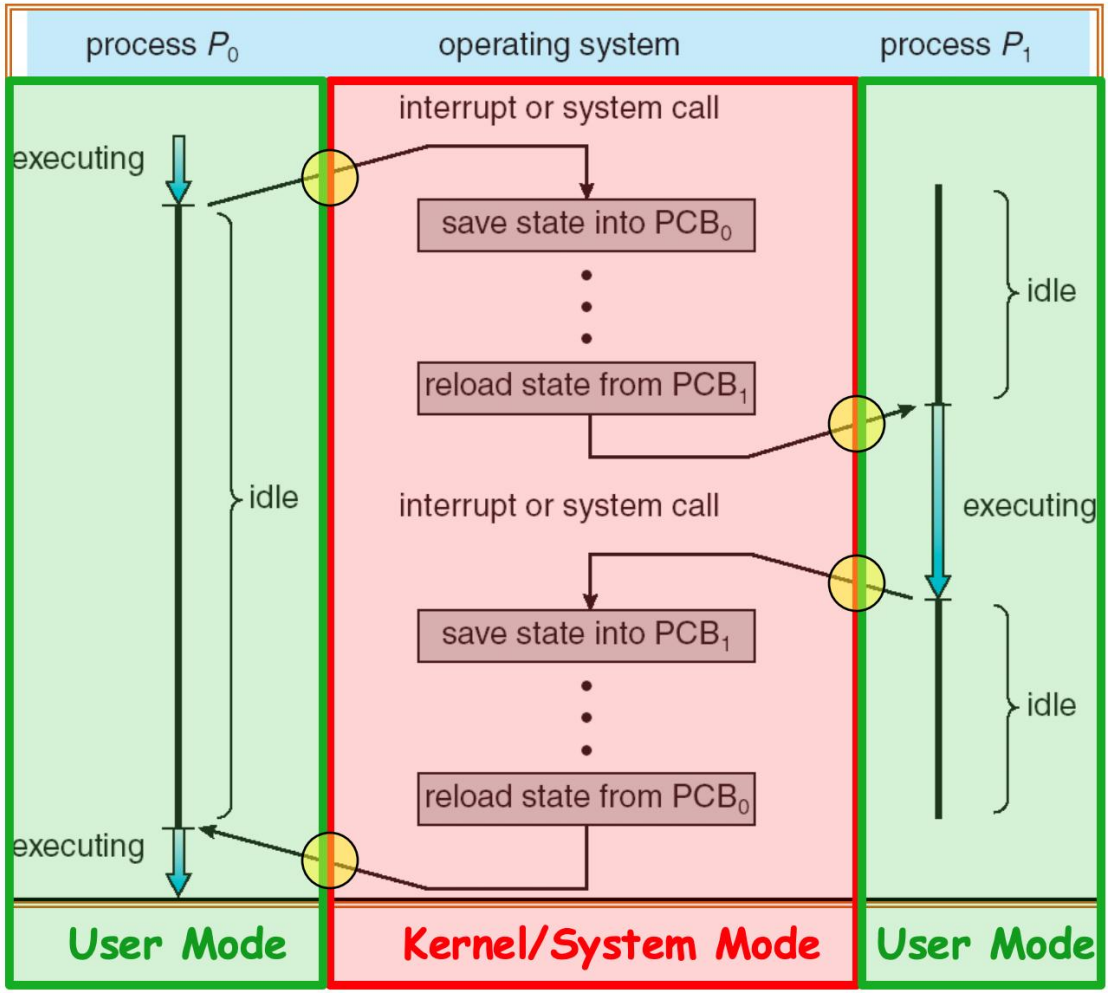
- *Dispatcher* thread
  - wait for a request
  - handoff the request
- *Worker* threads
  - process the request
    - disk I/O
  - respond the request
- "Many" requests at a time



User vs kernel modes



Context switching example



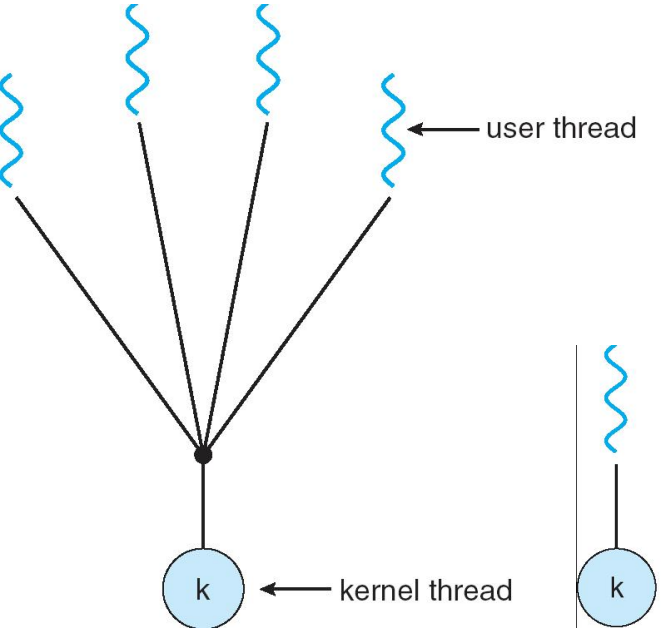
User vs kernel threads

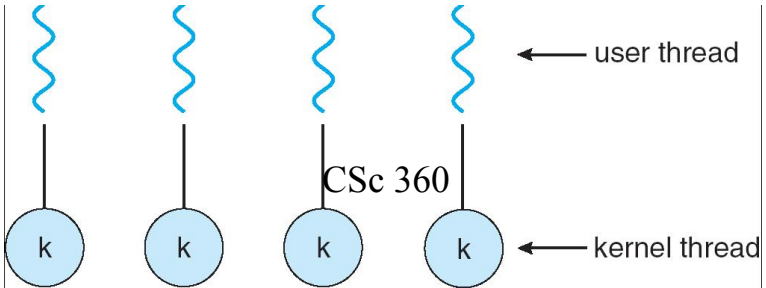
- User threads: e.g., pthread library
  - each process schedules its own threads
  - no context switch between these threads
  - a blocking call blocks the entire process
- Kernel threads: in almost all modern OS
  - kernel manages all threads
  - can pickup another thread if one blocks
- Hybrid approaches

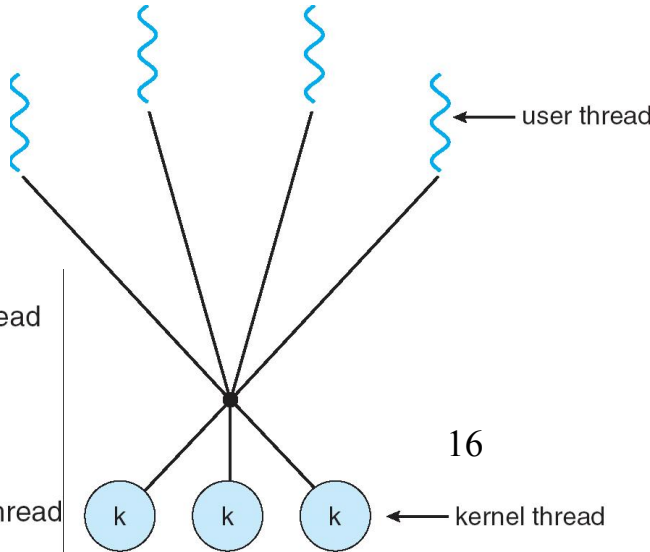
* Intel's hyper-threading?

Thread models

- User-kernel mapping
  - many-to-one: low cost, (lower) parallelism
  - one-to-one: high parallelism, (higher) cost
  - many-to-many: limited kernel threads







Threading issues

- When a new process is created
  - fork(), and usually then exec()
    - duplicate all threads or just the calling thread?
- When a signal to be delivered
  - signal: event notification to be handled
    - to all, some, or a specific thread?
- Thread pool
  - keep a pool of threads to be used
    - and reuse

This lecture so far

- Thread
  - a basic unit of CPU utilization
  - user vs kernel-level threads
    - thread models
  - issues with threading (and more later)
- Explore further
  - thread support in your favorite OS
    - user vs kernel, thread model?

Pthread library

- Create a thread
  - int **pthread_create** (thread, attributes, start_routine, arguments);
  - PC: start_routine(arguments);
  - default attributes: joinable and non-realtime
- Exit from a (created) thread
  - void **pthread_exit** (return_value);
  - cleanup handlers by **pthread_cleanup_push** ();
    - stack-like “reverse” execution order

thread vs process system calls/diagram on board

Pthread library: more

- Wait a target thread to exit: *synchronize*
  - int **pthread_join** (thread, return_value);
  - release resource allocated to the target thread
- Put a target thread in detached state
  - int **pthread_detach** (thread);
  - no other threads can “join” this one
    - no “pthread_attach”
  - resource released once the thread exits
    - thread can be created in detached state

Pthread: furthermore

- Cancel another thread
  - int **pthread_cancel** (thread);
  - calling thread: send a request
  - target thread: **pthread_setcancelstate** ();
    - ignore the request
    - terminate immediately
      - asynchronous cancellation
    - check periodically whether it should be cancelled
      - deferred cancellation

Example: producer-consumer

- Multi-process
  - shared memory solution
  - message passing solution
- Single-process, multi-thread

```
#include <pthread.h>
...

void *producer (void *args);
void *consumer (void *args);

typedef struct {...} queue;
```


* with a fifo queue instead of (ring) array

Main thread

```
queue *queueInit (void);
void queueDelete (queue *q);
void queueAdd (queue *q, int in);
void queueDel (queue *q, int *out);

int main ()
{
        queue *fifo;
        pthread_t pro, con;

        fifo = queueInit ();
        if (fifo ==  NULL) {
                fprintf (stderr, "main: Queue Init failed.\n");
                exit (1);
        }
        pthread_create (&pro, NULL, producer, fifo);
        pthread_create (&con, NULL, consumer, fifo);
        pthread_join (pro, NULL);
        pthread_join (con, NULL);
        queueDelete (fifo);

        return 0;
}
```

Producer thread

```
void *producer (void *q)
{
        queue *fifo;
        int i;

        fifo = (queue *)q;

        for (i = 0; i < LOOP; i++) {
                /* produce LOOP items, inserting them into
        * the “fifo” queue.
        */
                ...
        }
        return (NULL);
}
```

Consumer thread

```
void *consumer (void *q)
{
        queue *fifo;
        int i, d;

        fifo = (queue *)q;

        for (i = 0; i < LOOP; i++) {
                /* Consumer LOOP items from the
                 * “fifo” queue.
                 */
                ...
        }
        return (NULL);
}
```

The 2nd half of this lecture

- Pthread library
  - create and terminate threads
    - passing arguments: start_routine (arguments);
  - join and detach threads
    - synchronize
- Explore further

  Pthread tutorial

  https://computing.llnl.gov/tutorials/pthreads/

Next lecture

- Pthread
  - threads: sharing data in a process
    - read-write, write-write conflicts
  - mutex and condition variables

    https://computing.llnl.gov/tutorials/pthreads/