• Example – web server
  o Request comes in and page is sent out
  o Disk accesses take a long time, so assume a cache is used to store frequently requested pages
    ▪ Located in main memory
  o Algorithm
    ▪ Receive request
    ▪ Parse request
    ▪ Check cache to see if it is already in main memory
      ▪ If so, format and send reply
      ▪ If not, read page from disk
      ▪ Format and send reply
  o Single process model with blocking system calls
    ▪ Every step must be performed by the process (no threads)
    ▪ Disk request will cause the server to block
      ▪ Other requests could be coming in at that time
    ▪ Drawback – much fewer requests/second can be processed
    ▪ Benefit – very simple sequential program
  o Single process model with non-blocking system calls
    ▪ Every step must still be performed by the process (no threads)
    ▪ Disk read and network send will return immediately
      ▪ When the data is ready, a signal or interrupt will alert the process
    ▪ Server must record the state of every request in a table of some kind
      ▪ Necessary so that it will know what to do next when the signal/interrupt if triggered for that request
    ▪ Benefit – doesn’t block, so many more requests/second can be processed
    ▪ Drawback – Non-sequential programming requires us to maintain quite a bit of state, which is complicated
  o Can we get the ease of sequential programming with the performance gains of asynchronous I/O?
• Thread Model
  o Introduction
    ▪ Process groups related resources together
      ▪ Open files, child processes, pending alarms, signal handlers, etc.
    ▪ Process also consists of a thread of execution
      ▪ Program counter – next instruction to execute
      ▪ Registers – holds current working variables
      ▪ Stack – Execution history (one frame per procedure call)
You can separate the two in a process and support multiple threads of execution:

- A thread must execute in a process, but multiple threads can also exist.

Single CPU – switches back and forth among the threads as if they were processes.
- Multiple CPU – threads can be run in parallel.

Threads are not as independent as processes:

- Memory:
  - Every thread can access the same memory.
  - Can wipe out another thread’s stack.
- Made to cooperate, not fight.
  - No protection between threads.

Per-process items:

- Address space
- Global variables
- Open files
- Child processes
- Pending alarms
- Signals and signal handlers
- Accounting information
Per-thread items
- Program counter
- Registers
- Stack

Situations where threads are desirable
- Some applications have many different activities going on at once
  - One of the activities may block occasionally (or frequently)
  - Quasi-parallel threads can make it easier to conceptualize these activities
  - Same argument as processes, except threads share address space
- Easier to create/destroy than processes
  - Sometimes 10-100 times faster than creating a process
  - Because they are lighter weight (don’t require new memory)
  - When # of threads needed changes over time, can be useful
- Mitigates the effect of I/O bound processes
  - Threads won’t help when they are all CPU-bound
  - Processes with both significant I/O and CPU utilization can be helped by threads
- Useful on multicore and multiprocessor systems
  - Provides true parallelism within a single process

Back to web server example
- Dispatcher thread accepts requests
- Hands off request to one of a number of worker threads
  - Worker thread handles the blocking disk read
  - Other workers (and the dispatcher) can continue their work
- Benefit – code is sequential and broken into small finite jobs
- Dispatcher

```c
while (TRUE) {
    get_next_request(&buf);
    handoff_work(&buf);
}
```

- Worker

```c
while (TRUE) {
    wait_for_work(&buf)
    look_for_page_in_cache(&buf, &page);
    if (page_not_in_cache(page))
        read_page_from_disk(&buf, &page);
    return_page(&page);
}
```

- System Calls
  - Pthread_create – create a new thread
    - Will run a procedure of the following type:
      - void* thread_proc(void* args)
    - int pthread_create(pthread_t* thread, pthread_attr_t* attr, function, void* args)
    - Returns zero for success, -1 for failure
    - thread – contains thread ID
    - attr – attributes on how the thread will start
      - priority, joinable, etc.
    - args – passed to the thread function to be run
  - Pthread_exit – terminate calling thread
- void pthread_exit(void* retval)
  - retval is an integer akin to that returned by exit.

- Pthread_join – Wait for a specific thread to exit
  - int pthread_join(pthread_t thread, void** retval)
  - Returns zero for success, -1 for failure
  - thread – thread ID you want to wait for
  - retval – Copies the pointer value returned by pthread_exit into a local pointer
  - Blocking call unless the thread has already exited

- Pthread_yield – Release the CPU to let another thread run
  - Concept doesn’t exist in processes
  - int pthread_yield(void)
  - Moves itself to the bottom of its current priority queue

- Example

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

#define NUMBER_OF_THREADS 10

void *print_hello_world(void *tid)
{
    /* This function prints the thread's identifier and then exits. */
    printf("Hello World. Greetings from thread \%d, tid\);  
    pthread_exit(NULL); 
}

int main(int argc, char *argv[])
{
    /* The main program creates 10 threads and then exits. */
    pthread_t threads[NUMBER_OF_THREADS];
    int status, i;

    for(i=0; i< NUMBER_OF_THREADS; i++)
    {
        printf("Main here. Creating thread \%d, \%d\);  
        status = pthread_create(&threads[i], NULL, print_hello_world, (void *)i);

        if (status != 0) {
            printf("Oops. pthread_create returned error code \%d, status\);  
            exit(-1);
        }
    }
    exit(NULL);
}