Design and Engineering of Computer Systems

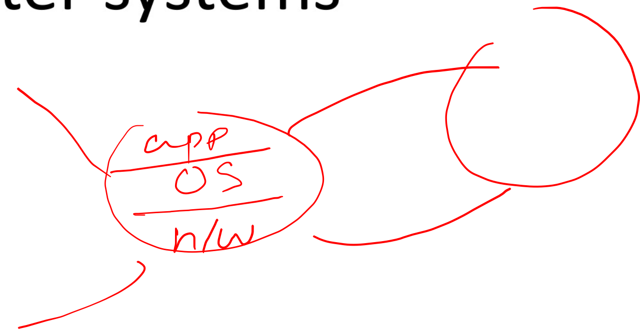
Lecture 26: Multithreaded application design

Mythili Vutukuru

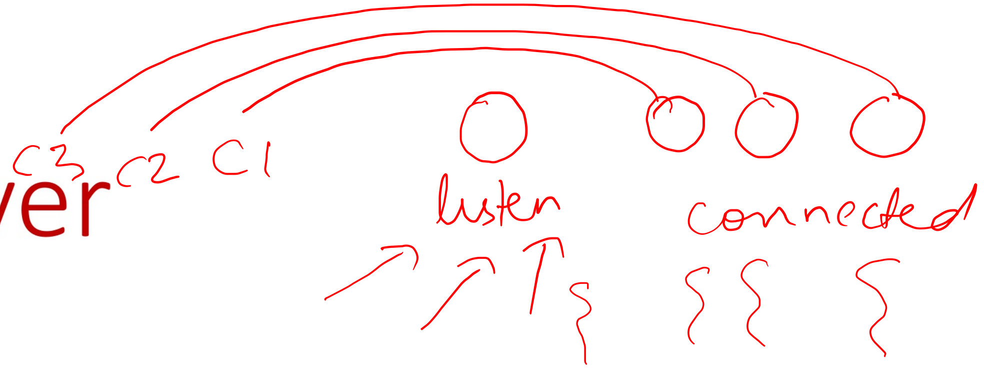
IIT Bombay

End-to-end system design

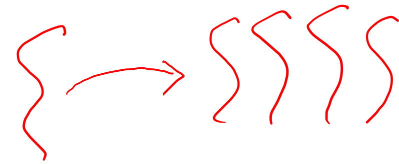
- What we have studied so far: building blocks for computer systems
  - Computer hardware, OS, syscall API to user space processes
  - How processes communicate over the network
- This week: end-to-end design of a computer system
- Computer systems and applications are not monolithic, composed of multiple components distributed across several machines
- Within a single machine, multiple threads or processes must coordinate with each other to implement functionality
  - This lecture: how multiple threads in a process work together
  - Next lecture: how multiple processes in a system work together



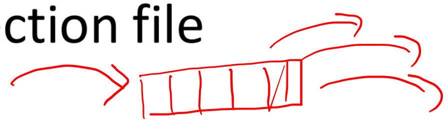
Multi-threaded server



- Consider the example of multi-threaded web/application/TCP server
  - Server has listen socket which listens for new connections
  - Server has multiple connected sockets, one for each connected client
  - One thread per connection design: main thread of server blocks on accept, per-client threads block on client reads and handle client requests
  - Use a pool of threads instead of creating/destroying new threads all the time



- Multi-threaded server using thread pool, or master-worker model
  - Main master thread of server accepts new connections, places new connection file descriptors (or requests) in a shared queue



  - Worker threads pick requests from the queue one by one, and service them
  - Mutual exclusion using locks when adding/removing requests from queue
- How does worker thread know when request has arrived in queue?
  - All worker threads constantly keep checking the queue all the time? (inefficient polling)

Condition variables

- Threading libraries provide various mechanisms for threads to synchronize and coordinate with each other efficiently
  - Example: Thread T1 does some task (e.g., add request to queue), only then thread T2 does something else (e.g., process request from queue)
  - Note that locks are not enough for such signaling
- Pthread library provides special variables called condition variables (CV)
  - A thread can call wait function on a CV, it will block and get added to a list of threads waiting on that CV
  - Another thread calls signal on a CV, one of the waiting threads gets ready to run again
- Example: use CV for "T1 does some work, only then T2 does something else"

T1

```
//do work
done = true
signal(cv)
```

T2

```
if(!done) wait(cv)
(T2 blocks)

(T2 resumes)
```

T1

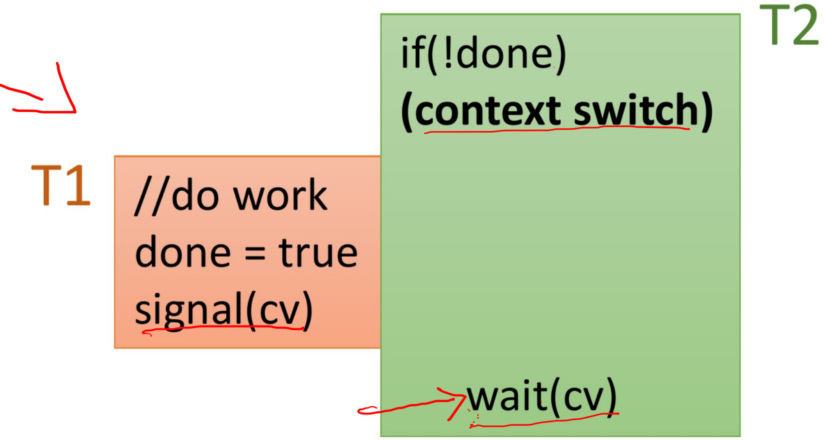
```
//do work
done = true
signal(cv)
```

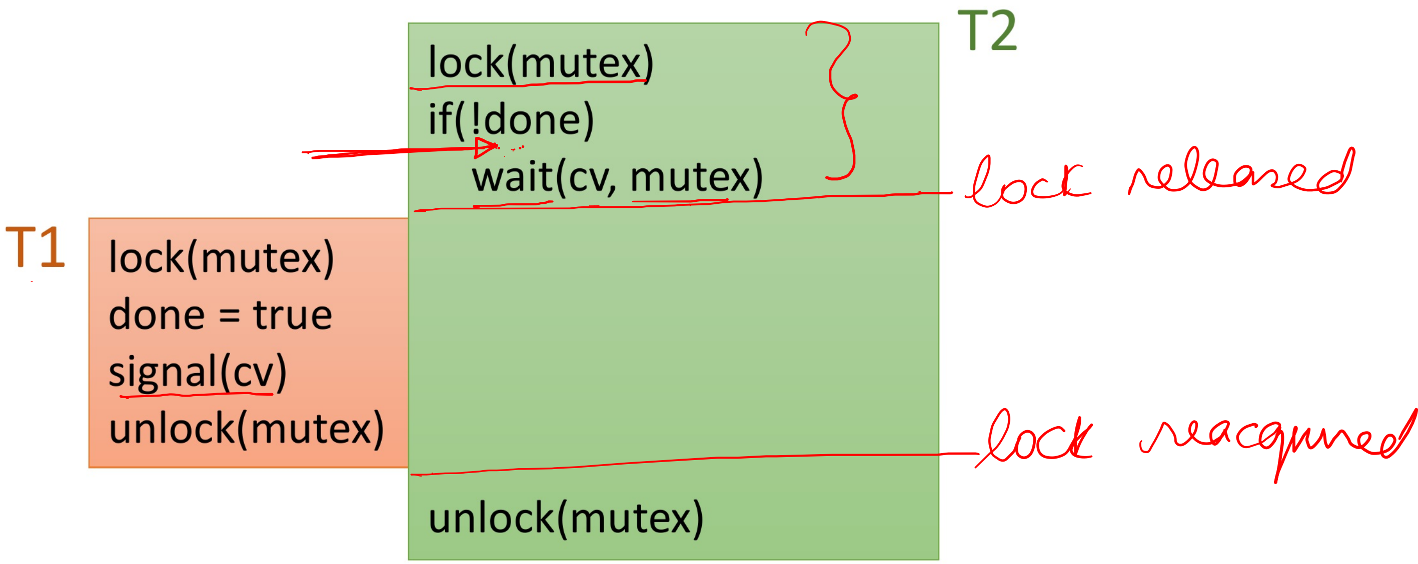
T2

```
if(!done) wait(cv)
//proceed, no wait
```

Atomicity in wait and signal

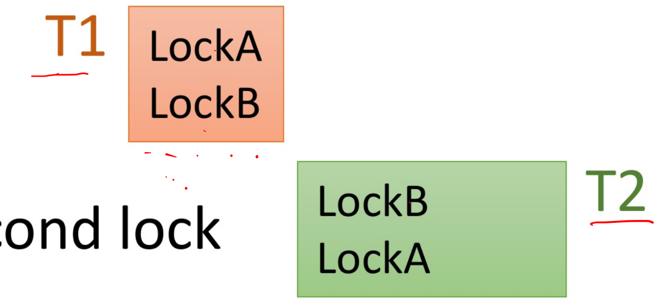
- Checking condition and waiting must be atomic, deadlock otherwise
  - Thread T2 checks condition is false, context switch just before blocking
  - Meanwhile T1 makes condition true, signal doesn't wake up anyone (none sleeping)
  - T2 resumes, goes to sleep forever (no one left to signal)
- Solution: use a lock/mutex to protect atomicity of sleeping
  - T2 holds a lock, checks condition, calls wait, lock released only after T2 is added to list of waiting processes (atomically check condition and sleep)
  - T1 holds lock when calling signal, ensures that signal cannot happen in between checking condition and waiting

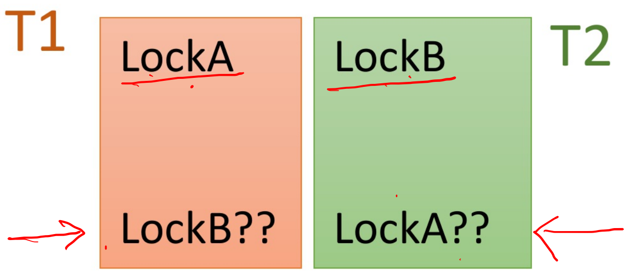




Deadlock

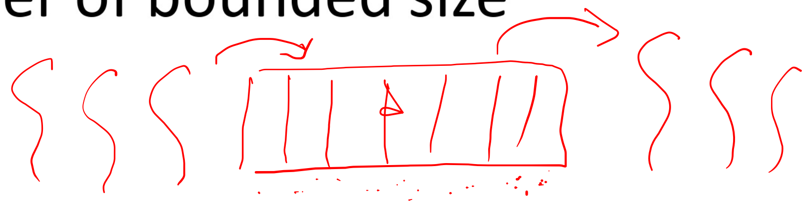
- Deadlock: threads are stuck in blocked state without making progress
  - Livelock: threads are running but doing wasted work, not making progress
- Example of deadlock: thread sleeps by calling wait on CV, no other thread calls signal, so thread sleeps forever
- Example: circular wait when acquiring multiple locks
  - T1 acquires LockA and LockB, T2 acquires LockB and LockA
  - T1 acquires LockA, T2 acquires LockB, each is waiting for second lock
  - Deadlock if executions interleave in some ways
- Techniques to avoid deadlocks
  - Acquire locks in same order across all threads of process
  - When sleeping, ensure someone will wake you up!





Producer-consumer with bounded buffer

- Producer and consumer threads, sharing data via a buffer of bounded size
  - Producers produce items, add into a shared buffer
  - Consumers consume item from shared buffer

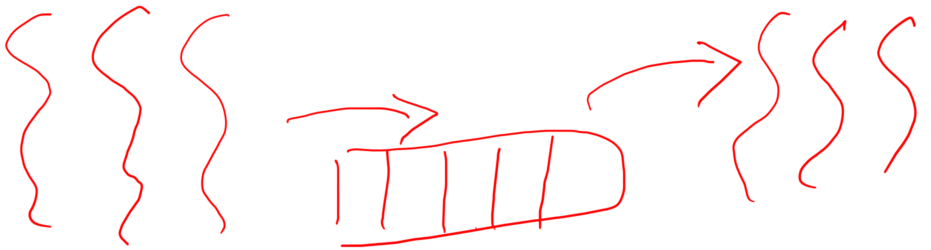


- What kind of coordination is needed between threads?
  - Producer thread cannot produce and waits if the buffer is full → Consumer signals after making space in the buffer
  - Consumer thread cannot consume and waits if the buffer is empty → Producer signals after producing items
  - Mutex/lock used while modifying shared buffer, in addition to two CVs

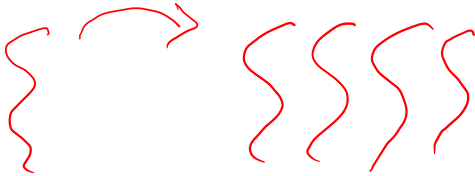
```
//Producer
lock(mutex)
if(no free space in buffer)
   wait(cv_buffer_full, mutex)
produce item, add to buffer
signal(cv_buffer_empty)
unlock(mutex)
```

```
//Consumer
lock(mutex)
if(no items in buffer)
   wait(cv_buffer_empty, mutex)
consume item from buffer
signal(cv_buffer_full)
unlock(mutex)
```

Multi-threaded server design

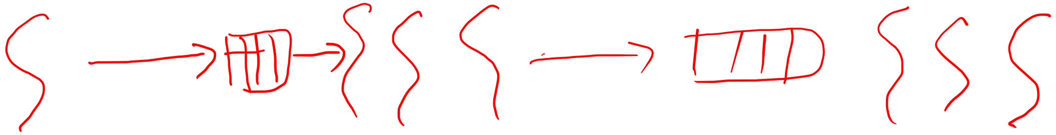


- Multi-threaded server with thread pool is a producer-consumer pattern
  - Master thread places requests in a shared queue
  - Worker threads take requests from queue and handle them as needed



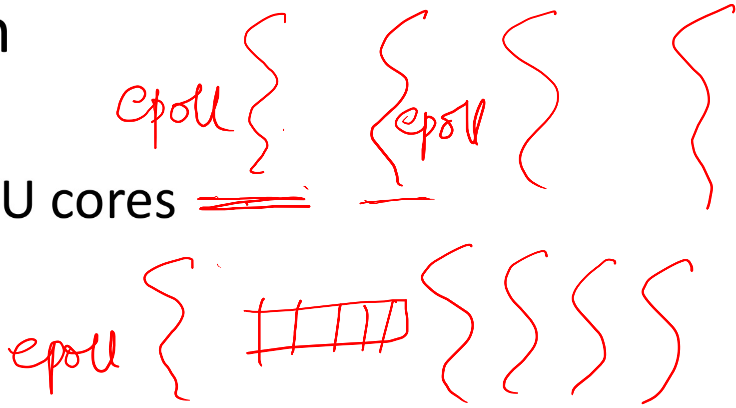
- How many threads in a thread pool? Optimum value to be tuned
  - Too few threads: queue builds up, clients not served on time, server CPU cores under-utilized due to not enough parallelism
  - Too many threads: unnecessary overhead of context switches and memory use
- How is request processing handled by a worker thread?
  - Run-to-completion: one worker thread handles client request from beginning to end, blocking across multiple steps

  - Pipeline: worker thread handles one part of request, places it in queue for next stage



Event-driven multi-threaded server

- What if event-driven API (e.g., epoll) used to handle multiple concurrent clients at server? No need for one thread per connection
- However, multiple threads still needed because
  - Single threaded epoll server cannot effectively use multiple CPU cores
  - Single threaded epoll server cannot do blocking disk I/O
- Event-driven multi-threaded server design choices
  - Multiple threads on multiple cores, each using epoll to manage multiple clients
  - One master thread performs epoll, reads client requests, hands over requests to thread pool of worker threads (which can block for disk I/O)
- Whether using blocking APIs or event-driven APIs, servers running on multicore systems usually have multiple threads
  - Understanding mechanisms for thread coordination (like CVs) is important



Summary

- In this lecture
  - How to build multi-threaded servers
  - Synchronization across threads using condition variables
  - Concurrency bugs like deadlocks
- Programming exercise: extend a simple client-server socket program by adding a thread pool at the server. Use pthreads condition variables to coordinate across the master and worker threads.