CS 347M (Operating Systems Minor)

Spring 2022

Lecture 14: Condition variables

Mythili Vutukuru

CSE, IIT Bombay

Recap: Socket communication

- Client and server applications on the Internet communicate with each other using sockets, e.g., web browser and web server
  - Server opens socket on a well known address and starts listening
  - Client opens a socket and connects to server's socket
  - Client and server exchange requests and responses

Client

```
sockfd = socket(..)
connect(sockfd, server_sockaddr, ..)
n = send(sockfd, req_buf, req_len, ..)
n = recv(sockfd, resp_buf, resp_len, ..)
```

Server

```
sockfd = socket(..)
bind(sockfd, server_address)
listen(sockfd, ..)
newsockfd = accept(sockfd, ..)
n = recv(newsockfd, req_buf, req_len, ..)
n = send(newsockfd, resp_buf, resp_len, ..)
```

Multi-threaded applications

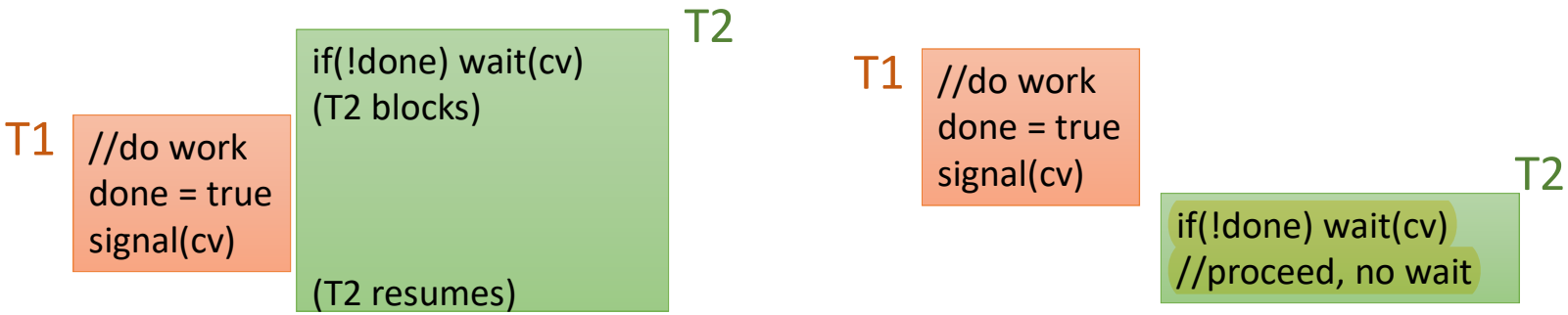
- Real-life applications do multiple things concurrently
  - Server has to listen for and accept new connections from new clients
  - Server has to handle requests coming in from existing clients
  - These various operations may block (e.g., read blocks till data arrives)
- Real-life applications have multi-threaded designs, e.g., master-worker
  - Main master thread of server accepts new connections, places new connections 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 check the queue all the time? (inefficient polling)

What we need: wait and signal mechanism

- Locks allow one type of synchronization between threads – mutual exclusion when accessing critical sections
- Another common requirement in multi-threaded applications – waiting for events and signaling when event occurs
  - E.g., Worker thread wants to run only after master thread has placed a new request in the shared queue
  - E.g., Thread T2 wants to run only after T1 has finished some task (T1→T2)
- Naive solution: T2 keeps checking periodically if T1 is done
  - Wastes CPU cycles, inefficient
  - Need a new synchronization primitive to wait for an event

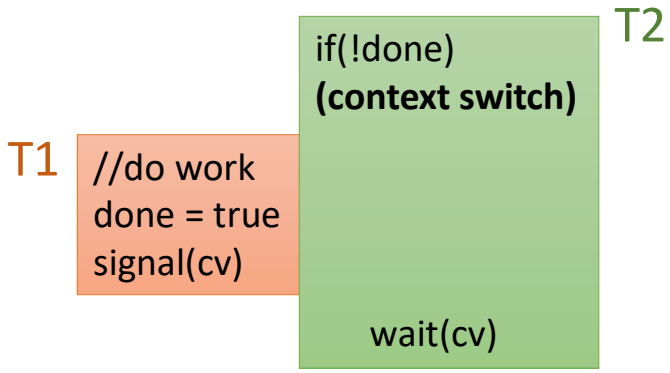
Condition variables

- Pthread library provides special variables called condition variables (CV)
  - A thread calls wait function on a CV, it is blocked and gets 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, will be scheduled in the future (no immediate context switch)
- Example: we want T2 to run only after T1 does its work (T1→T2)
  - T1 does its work and calls signal
  - T2 checks if work is done, and calls wait if work is not done



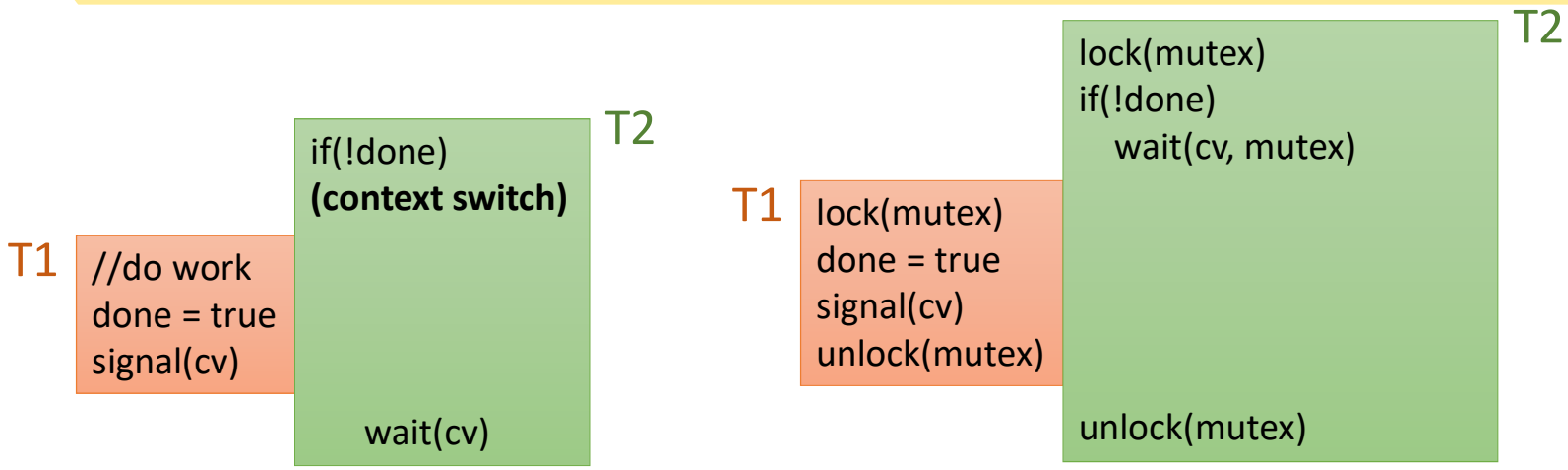
Atomicity in wait and signal (1)

- 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, calls signal. But signal doesn't wake up anyone (none sleeping yet)
  - T2 resumes, goes to sleep forever (no one will signal again)
- This is called missed wakeup problem: how to fix?



Atomicity in wait and signal (2)

- 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 (ensures atomicity of checking condition and sleeping)
  - T1 acquires **same** lock before calling signal, ensuring that signal cannot happen in between checking condition and waiting
  - Pthread CV implementation releases lock during wait, reacquires on wakeup



Guidelines for using condition variables

- Use the same lock for wait and signal (maybe for other variables too)
- Before calling wait, confirm that the condition is indeed false
  - T2 must check "done" variable before calling wait (what if T1 has already run?)
- Signal broadcast wakes up all threads while signal wakes up any one
- Good habit to check condition with "while" loop and not "if"
  - To avoid corner cases of thread being woken up even when condition not true (may be an issue with some implementations)

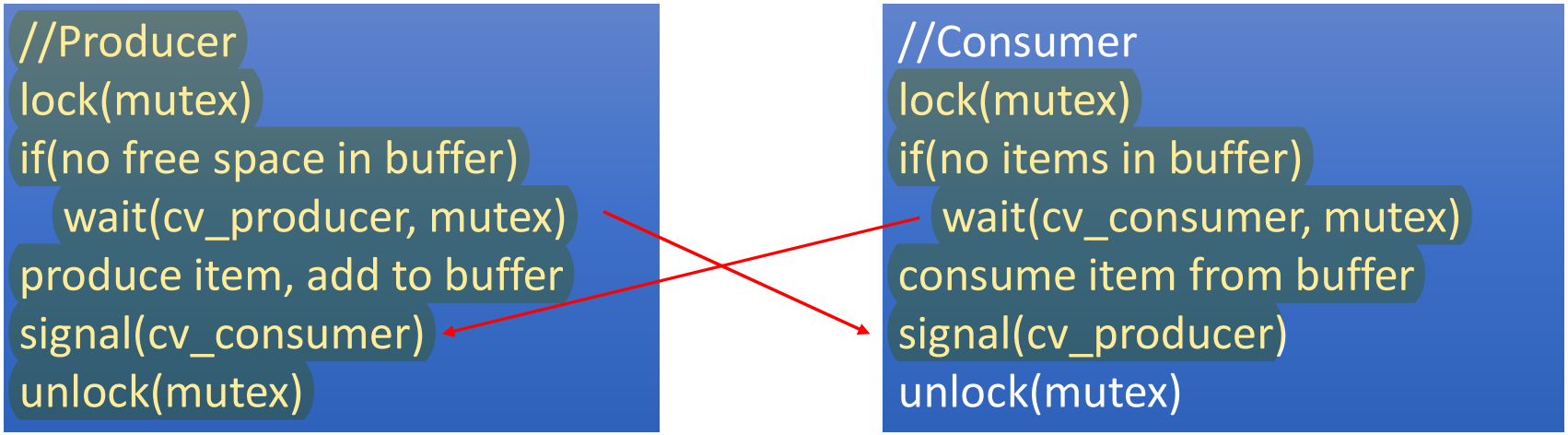
```
if(condition)
  wait(condvar)
//small chance that condition may be false when wait returns

while(condition)
  wait(condvar)
//condition guaranteed to be true since we check in while-loop
```

Example: Producer-consumer problem (1)

- 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
- Example: in a multi-threaded server
  - Master thread accepts requests from the network and puts them in a queue
  - Worker threads fetch requests from this queue and process them
- What kind of coordination is needed between threads?
  - Producer thread produces and places items into buffer, waits if the buffer is full → Consumer signals after making space in the buffer
  - Consumer thread consumes items from buffer, waits if the buffer is empty → Producer signals after producing items

Example: Producer-consumer problem (2)

- Solution using condition variables
  - Mutex/lock used while modifying shared buffer
  - Two CVs: one for producers to wait, and one for consumers to wait



Example: Batched processing (1)

- Two kinds of threads in an application
  - Request threads, each containing an application request
  - Batch processor thread processes N requests at a time in a batch
- What kind of synchronization do we need?
  - Batch processing thread must wait until N requests arrive, then start batch
  - Request thread must wait until batch starts, then get processed and finish
- Example: suppose Covid-19 vaccination vial has 10 doses. Nurse waits for 10 patients to arrive, then opens the vial and vaccinates all 10

Example: Batched processing (2)

- Solution using two CVs: one for requests to wait, one for batch processor to wait
  - Other integer and Boolean variables, mutex/lock for atomicity

```
//Request thread
lock(mutex)
count++
if(count == N)
   signal(cv_batch_processor)
while(not batch_started)
   wait(cv_request, mutex)
unlock(mutex)
```

```
//Batch processor thread
lock(mutex)
while(count < N)
   wait(cv_batch_processor, mutex)
batch_started = true
signal_broadcast(cv_request)
unlock(mutex)
```

Example: Batched processing (3)

- What is wrong with this solution?
  - Nth request thread calls wait before invoking signal to wake up batch processor
  - Batch processor never wakes up, all threads will sleep forever
  - Before you sleep, ensure that the signaling code can run in future

```
//Request thread
lock(mutex)
count++
while(not batch_started)
    wait(cv_request, mutex)
if(count == N)
    signal(cv_batch_processor)
unlock(mutex)
```

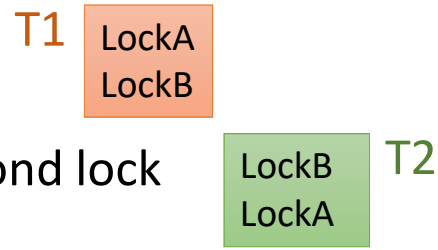
```
//Batch processor thread
lock(mutex)
while(count < N)
    wait(cv_batch_processor, mutex)
batch_started = true
signal_broadcast(cv_request)
unlock(mutex)
```

Synchronization patterns using CVs

- Many examples in the practice problems
  - Scenario describing multiple threads/entities and how they should interact and coordinate with each other
  - Toy examples modelled after real world application design patterns
- How to write code with correct synchronization
  - Identify when each entity should wait and write the suitable waiting code
  - For each wait, figure out how the signaling will happen and write the code
  - Ensure that signaling path in the code is not blocked in any way, e.g., signal others first before calling wait and going to sleep
  - Update all extra variables (counts, flags) in the solution correctly
  - Run through your code in a few different scenarios and different order of execution of threads to convince yourself that it works correctly

Watch out for deadlocks

- Deadlock: threads are stuck in blocked state without making progress
- Example: 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!