Synchronization Tools

2023/24 COMP3230B

Contents

- mutual exclusion – mutex lock
- Using atomic instructions to build spin-waiting mutex lock
- Condition variables – a synchronization tool
- Semaphores – another tool that can serve for locking and/or synchronization

Related Learning Outcomes

- ILO 2c – describe the principles and **techniques** used by OS to support concurrency and synchronization control.
- ILO 4 – **[Practicability]** demonstrate knowledge in applying system software and tools available in modern operating system

Readings & References

- Required Readings
  - Chapter 28 – **Locks** (except sections 28.10, 28.11, 28.15, 28.16)
    - http://pages.cs.wisc.edu/~remzi/OSTEP/threads-locks.pdf
  - Chapter 30 – **Condition Variables**
    - http://pages.cs.wisc.edu/~remzi/OSTEP/threads-cv.pdf
  - Chapter 31 – **Semaphores**
    - http://pages.cs.wisc.edu/~remzi/OSTEP/threads-sema.pdf

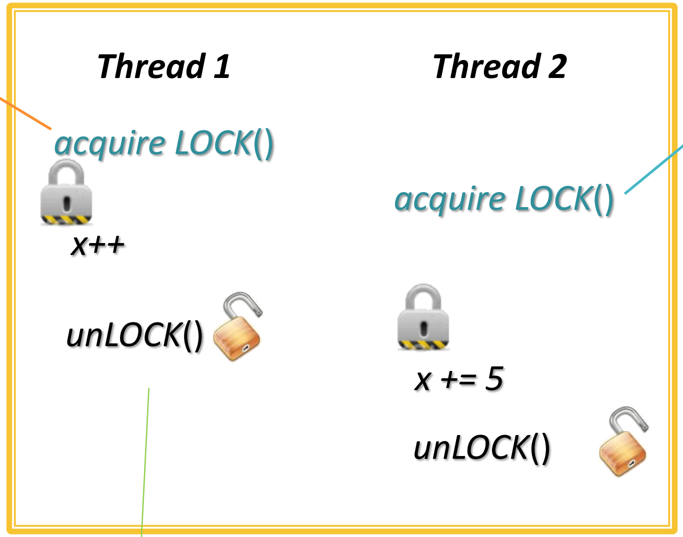
Mutual Exclusion

- We need a mechanism to inform OS that a thread is going to enter its critical section
- Here comes the concept of using a “lock”
  - A lock is a data structure used for **indicating** the start and end of **a critical section**
    - Indicate shared data is about to be accessed and **ask for the system** to provide necessary protection

Mutual Exclusion – Lock

- A lock is either available (free or not locked) or held (locked or acquired)

A thread wants to access a shared item, it tries to acquire a lock



If the lock is held by another thread, this thread will not return and wait until the lock changes to free; **then** it acquires the lock and enters critical section

If no other thread holds the lock, this thread gets the lock (becomes the **owner**) and enters critical section

The owner calls unlock( ) to free the lock when exits critical section

Building Locks

- The main cause of the indeterministic outcome is ***uncontrolled scheduling***
  - Can we avoid being preempted when the thread is in critical section?

/ lock( )*/*
disable interrupts

/ unlock( )*/*
enable interrupts

- **Disabling interrupts**
  - Prevents current executing thread from being preempted, as without interrupts, scheduler will not be invoked; thus, no other threads will be able to turn to running state
- Works **only** on systems with single core
- Being used rarely, mostly used in kernel **under privilege mode**

Atomic Instructions

- Modern machines provide special atomic hardware instructions
- **Test-and-Set** Instruction
  - It enables us to test the old value ***while simultaneously set*** the variable to a new value
    - The instruction ***returns the old value*** and **simultaneously updates the variable** to **new value** in an **atomic way**

Guarantee execution till end without interruption

```
int TestAndSet(int *ptr, int new) {
    int old = *ptr;         //get current value
    *ptr = new;             //set to new value
    return old;    //return old value
}
```

- **Compare-and-Swap** Instruction
  - Test whether the content of a variable is equal to expected; if so, update the variable to a new value; otherwise, do nothing
    - The instruction **always returns the current value** of the variable

Guarantee execution till end without interruption

```
void CompareAndSwap(int *ptr, int expected, int new)
{
    int temp = *ptr;         //get current value
    if (temp == expected)
        *ptr = new;
    return temp;
}
```

Test-and-Set

- Implementing Mutual Exclusion
  - Based on **checking** of the value of a shared variable to decide whether a process can enter a Critical Section
    - Uses **busy waiting** (spin-wait or spin-lock) to test whether it can enter the Critical Section

If another thread is in critical section, flag is 1; in that case, TestAndSet( ) will return 1

If no thread is in critical section, flag is 0; in that case, TestAndSet( ) will return 0 and will atomically modify flag to 1

```
int flag = 0;        //when flag = 0, lock is free

while (TestAndSet(&flag, 1) == 1);      //lock( )

// into the critical section

flag = 0;    //unlock( )
```

Test-and-Set

- With the guarantee of atomic update of a shared variable
  - This guarantees mutual exclusion as only one thread can set the shared flag to 1, others have to spin-wait for the flag to turn back to 0
- Spin-wait / Busy-waiting
  - However, the processor is essentially doing nothing but just executes the while() loop
  - Wastes significant processor time
    - those threads may just spin-waiting until the time quantum expires
- Spin-waiting cannot avoid indefinite postponement; thus, do not provide fairness guarantee
  - e.g., when two threads T1 & T2 contend for entering to critical sections, because of uncontrolled scheduling, it is possible that T1 always be the lucky one & T2 always finds that the flag is always 1

CompareAndSwap

- Example: Intel IA-32 and IA-64 contain an CMPXCHG instruction
- Implementing Mutual Exclusion
  - Again, using spin-waiting

```
int flag = 0;             //lock is free

                T1

  while (CompareAndSwap(&flag, 0, 1));

       //in critical section

    flag = 0;   //free the lock

       //non-critical region
```

flag=0

flag=1

```
                T2

  while (CompareAndSwap(&flag, 0, 1));

    //in critical section

    flag = 0;   //free the lock

       //non-critical region
```

flag=1

flag=0

flag=1

flag=0

MutEx – Hardware Instructions

- Atomic instructions
  - Applicable to any number of processes/threads on either a single processor or multiple processors sharing main memory
- Using Machine instructions cannot provide a good mutual exclusion solution on its alone
  - For example, possibility of having indefinite postponement if more than one thread is waiting
- Unfortunately, Spin-waiting consumes processor time

- The Crux
  - How can we develop a lock that does not needlessly waste time spinning on the CPU as well as maintain fairness?
- The answer is: we need OS support.

Pthread Lock

- In POSIX library, the lock data type is called mutex
- mutex variables must be declared with type
  - pthread_mutex_t
- Must be initialized before they can be used

- Two operations on mutex variables
  - pthread_mutex_lock() and pthread_mutex_unlock()
    - **OS blocks the calling thread** if the request lock is not available
      - So the thread transits to Blocked STATE
- When finished using a mutex, deletes it with
  - pthread_mutex_destroy(&lock);

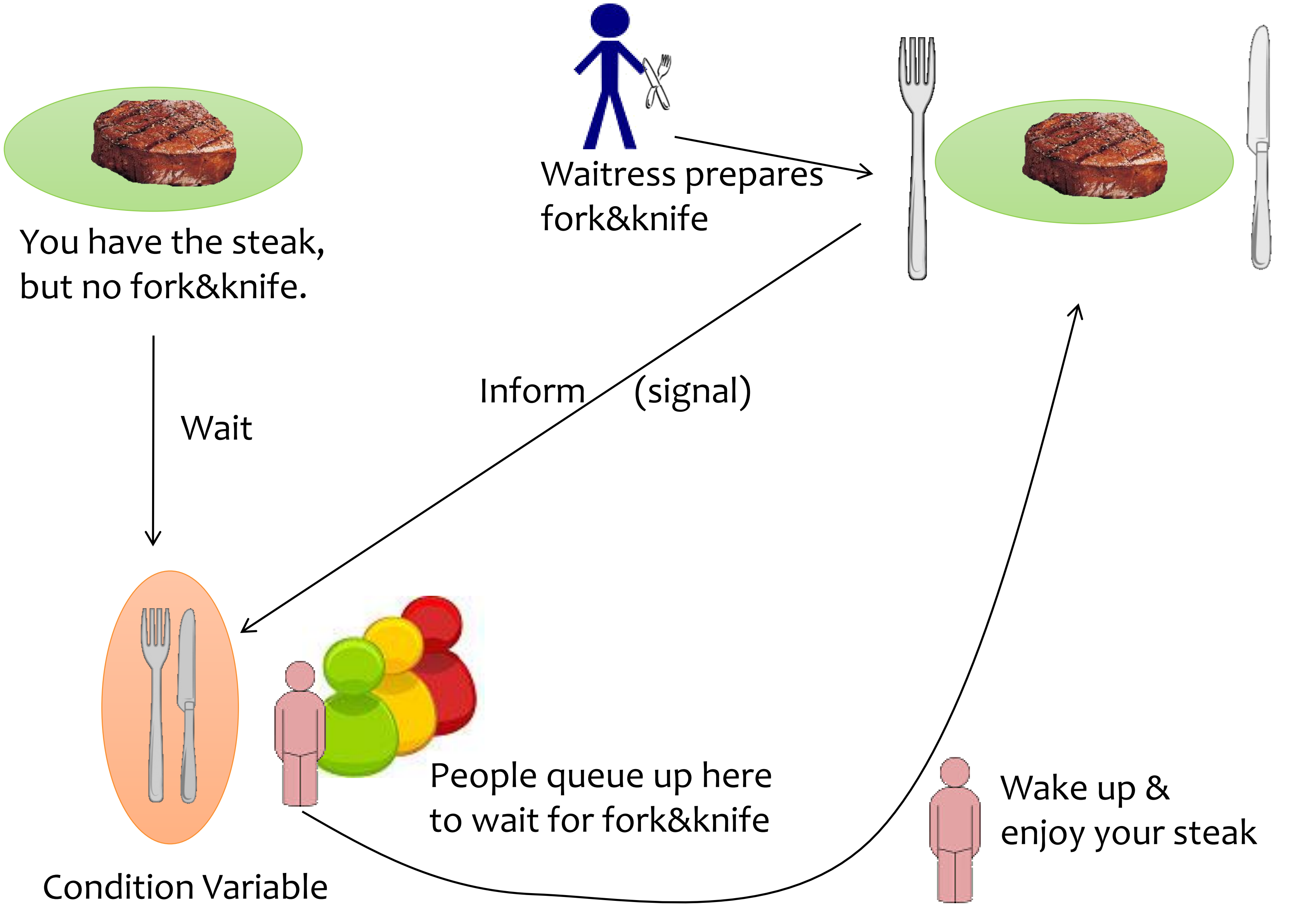
Control Synchronization

- Another typical interaction between threads is the control synchronization
  - a thread (A) should perform some action only after some other threads have performed specific actions (or have detected an event/condition)
  - **Thread A has to be waited and some other thread has to notify A** about the occurrence of the event/condition

- A simple approach – Thread A **just spins until the condition** becomes true
  - this is inefficient and wastes CPU cycles.

Condition Variables

- A data type explicitly designed to support synchronization between threads (without wasting CPU cycles)
  - it has a queue that threads can wait over there when some condition/event is not met
  - some other thread, once detects the condition/event, inform (signal) those waiting threads and wake one of those waiting threads
  - when using condition variables, each condition variable can be associated to a distinct condition/event


Waitress prepares
fork&knife
You have the steak,
but no fork&knife.
Wait
Inform
(signal)
People queue up here
to wait for fork&knife
Wake up &
enjoy your steak
Condition Variable

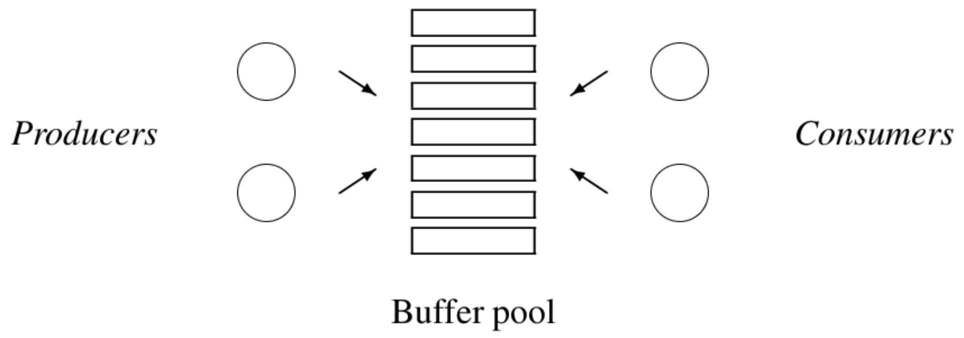
Pthread Condition Variables

- Must be declared with type pthread_cond_t, and must be initialized before they can be used

- To wait for a condition
  - pthread_cond_wait()
    - When call this function, the calling thread is BLOCKED until wake up by other using pthread_cond_signal()/broadcast()
- To inform others that the condition/event has happened
  - pthread_cond_signal()
    - Wake up one thread which is waiting on the condition variable
  - Pthread_cond_broadcast()
    - Wake up all threads currently blocked on the condition variable
- Finish using a condition variable
  - pthread_cond_destroy(&cond);

Producer-consumer problem

- Classical synchronization problem, also known as Bounded-Buffer Problem
  - The system has a **finite buffer pool** shared by one or more producers and one or more consumers
    - Each buffer is capable of holding one unit of information
  - Producers produce data items and wish to place them in buffers
    - A producer thread produces a data item and places it into the next available buffer in the buffer pool
  - Consumers take out data items from buffer and consume/process the data item in some way
    - A consumer thread consumes a data item by removing it out of the buffer pool



Producer-consumer problem

- Synchronization requirements
  - Buffer pool is shared resource, producers and consumers need to use some method to coordinate the access to the pool
  - A producer must not overwrite a buffer when buffer pool is full
  - A consumer must not consume an empty buffer when buffer pool is empty
  - Mutual exclusion
  - Information must be consumed in FIFO order

A Solution to Producer-consumer problem

To wait for buffer pool becomes NOT FULL

```
buffer[sizeofbuffer];
mutex_t mv;
cond_t notFULL, notEMPTY;
count = 0;

Producer( ){
        while (1)
        {
                d = generating_data();
                lock(mv);                                   //try acquire the mutex lock
                while (count == sizeofbuffer)               //while buffer pool is full
                        cond_wait(notFULL, mv);             //wait for free space
                append(d, buffer);                          //update the count
                cond_signal(notEMPTY);                      //data is available in buffer
                unlock(mv);                                 //release the lock
                remaining_work();
        }
```

To wait for data/job in buffer pool

- Before waiting, it releases the mv lock.
- Then queues in the cond queue.
- Upon waking up, it gets back the mv lock before return

Buffer is definitely NOT FULL

Buffer pool is definitely NOT EMPTY.
Inform waiting threads if any

A Solution to Producer-consumer problem

Buffer is definitely NOT EMPTY

```
Consumer( ) {
        while (1)
        {
                lock(mv);                                 //try acquire the mutex lock
                while (count == 0)                        //while buffer pool is empty
                        cond_wait(notEMPTY, mv);          //wait for data item
                d = take(buffer);                         //update the count
                cond_signal(notFULL);                     //one more free buffer space
                unlock(mv);                               //release the lock
                consume_data(d);
                remaining_work();
        }
}
```

Buffer pool is definitely NOT FULL.
Inform waiting threads if any

- Before waiting, it releases the mv lock.
- Then queues in the cond queue.
- Upon waking up, it gets back the mv lock before return

Semaphores

- Another synchronization tool that can serve as **mutex locks** or use for **control synchronization**

- A semaphore is an object with an internal protected integer variable
- Must be declared with type **sem_t**

- The integer value stored in the semaphore determines its behavior

- To initialize semaphore, **sem_t** *k*, to a value of *x*
  - sem_init(&*k*, 0, *x*);

Semaphores (2)

- The protect integer value **can only be** accessed via two atomic operations
  - sem_wait(&*k*) and sem_post(&*k*)
  - Also called P() (or wait or down) and V() (or signal or up), respectively

sem_wait(&*k*)

```
sem_wait(&k) {
    decrement the value of semaphore k by 1

    if value of semaphore k < 0
        place the thread in k's waiting queue
}
```

sem_post(&*k*)

```
sem_post(&k) {
    increment the value of semaphore k by 1

    if any threads are waiting on k's queue
        wake up one thread
}
```

MutEx – Binary Semaphores

- Semaphores that work like a mutex lock
- Binary – semaphore can have the value **one or zero only**
  - allow only one thread in its critical section at once
  - sem_t *lock* is **initially set to one only**

```
sem_t lock;
sem_init(&lock, 0, 1);

            Thread 1

while (1) {                    lock = 1
    sem_wait(&lock);           lock = 0

    //in critical section

    sem_post(&lock);

    //other code fragment
}
```

```
            Thread 2

while (1) {

    sem_wait(&lock);      lock = 0 (q=1)


    //in critical section  lock = 0 (q=1)

    sem_post(&lock);

    //other code fragment
}
```

```
            Thread 3

while (1) {

    sem_wait(&lock);      lock = 0 (q=2)



    //in critical section  lock = 0

    sem_post(&lock);      lock = 1
```

Readers-Writers Problem

- Another classic synchronization problem
- Consists of a set of threads accessing some shared data
  - Readers – threads that only reads the data
  - Writers – threads that modifies the data
- Typical example – Database access, linked list update and lookup
- Synchronization requirements
  - Many readers can perform reading concurrently
  - Reading is prohibited while a writer is updating
  - Only one writer can perform updating at any time

A Solution using Binary Semaphore

```
Semaphore semMutex = 1;

Semaphore readMutex = 1;

int readcount = 0;
```

```
Reader() {
    StartRead();
    do_reading();
    EndRead();
}
```

```
Writer() {
    StartWrite();
    do_writing();
    EndWrite();
}
```

```
StartRead() {
    P(readMutex); //Get the readMutex
    readcount++; //Count the number of readers
    if (readcount == 1) //If is the first reader
        P(semMutex); //Get the access right
    V(readMutex); //Release, others can update readcount
}

EndRead() {
    P(readMutex); //Finish reading, try decrement readcount
    readcount--;
    if (readcount == 0) //If is last reader, free access semaphore
        V(semMutex);
    V(readMutex);
}
```

```
StartWrite() {
    P(semMutex);
}
```

```
EndWrite() {
    V(semMutex);
}
```

General Semaphores

- Also known as **Counting Semaphores**
- Can be used to control access to a pool of identical resources
  - Initialized with sem_t *pool* to the resource count
  - Decrement the semaphore when taking resource from pool – sem_wait(&*pool*) operation
  - Increment the semaphore when returning it to pool – sem_post(&*pool*) operation
  - If no resources are available, thread is blocked until a resource becomes available

Synchronization bet. threads

- Semaphores can be used to notify other threads that specific event/condition have occurred/reached

- Example: Another example of common synchronization problems is **order of execution** of operations of different threads
  - Thread1 wants to wait for both Thread2 and Thread3 to finish their tasks before its operation

```
sem_t order;
sem_init(&order, 0, 0);    //initialize to 0
```

Thread1	Thread2	Thread3
:	:	:
:	:	:
sem_wait(&order);	computation2( );	computation3( );
sem_wait(&order);	sem_post(&order);	sem_post(&order);
computation1( );	:	:
:	:	:

Summary

- To avoid race condition, such that only one thread can be in critical section, we need to use mutual exclusion primitives, such as, mutex lock or binary semaphore, as a guard for accessing/modifying the shared data
- For spin-wait locks, in uniprocessor, because of the busy waiting, performance overhead can be quite painful
- By the use of condition variables, thread can have a non-busy waiting mechanism, by put itself into sleep and wait for other thread to wake it when the desire condition is met
- Semaphores are a powerful and flexible primitive that support both synchronization and mutual execution