# CSc 360
# Operating Systems
# **Process Synchronization**

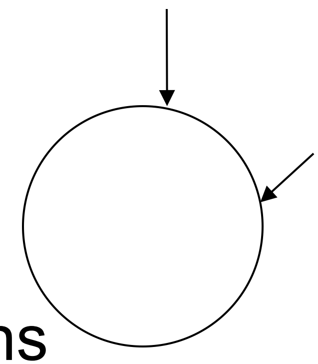
Wenjun Yang

**Fall 2025**

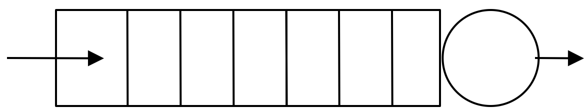
# The need for synchronization

- Multiprogramming
  - multi-process
    - process communication
    - shared memory or message passing
  - multi-thread
- CPU scheduling
- Cooperating processes/threads
  - e.g., the “producer-consumer” problem
    - cannot consume the things not produced yet

# The producer-consumer problem



- Solutions so far
  - bounded buffer
    - in, out variables; full, empty conditions
    - N buffer space, N-1 utilized at most
  - first-in-first-out queue
    - FIFO variable



- A simpler solution
  - to fully utilize the circular buffer
  - use a "counter" variable

# The “counter” solution

```
while (true) {
     /*  produce an item and put in nextProduced  */
     while (count == BUFFER_SIZE) ; // do nothing
     buffer [in] = nextProduced;
     in = (in + 1) % BUFFER_SIZE;
     count++;
}
```

Producer

```
while (true)  {
     while (count == 0) ; // do nothing
     nextConsumed =  buffer[out];
     out = (out + 1) % BUFFER_SIZE;
     count--;
     /*  consume the item in nextConsumed */
}
```

Consumer



Q: what's the problem?

# Race condition

- E.g., increment a counter (shared variable)
  - read the counter (from memory)
  - increment by one (at CPU)
  - write the counter
- How about two threads?
  - *sharing* only one counter

    e.g., counter=5 initially

    - non-deterministic result: $R_1W_1R_2W_2$; $R_1R_2W_1W_2$
- "There is something not to be (always) shared"



Q: how about count++ and count-- concurrently?

# Critical section

- Critical section
  - code section accessing shared data
  - only one thread executing in critical section
    - only one thread accessing the shared data: serialize
  - choose the right (size of) critical section!
- Approach: exclusion (lock)
  - if locked, wait!
  - if not lock, lock (and later, unlock)

# Properties of “solutions”

- Mutual exclusion
  – no more than one process in the critical section
- Making progress
  – if no process in the critical section, one can in
- Bounded waiting
  – for processes that want to get in the critical section, their waiting time is bounded

# Problem formulation

- Only 2 processes, $P_0$ and $P_1$
- General structure of process $P_i$ (other process $P_j$)

**do** {
  *entry section*
    /* critical section */
  *exit section*
    /* remainder section */
} **while (1)**;

- Processes may ***share*** some common variables to synchronize their actions
  - do not get into the loop!

# Algorithm 1

- Shared variables
  - int turn; // initially **turn = 0**
  - turn == i: $P_i$ can enter its critical section
- Process $P_i$

```
do {
        while (turn != i) ;   // wait
                /* critical section */
        turn = j;
                /* remainder section */
} while (1);
```

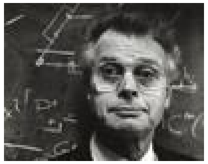
- Fate on other's hands: any problems?

# Algorithm 2

- Shared variables
  - **boolean flag[2]**;
    initially **flag [0] = flag [1] = false.**
  - **flag [i] = true** : $P_i$ ready to enter its critical section
- Process $P_i$

```
do {
        flag[i] := true;
        while (flag[j]) ;    // wait
                /* critical section */
        flag [i] = false;
                /* remainder section */
} while (1);
```

- Fight for access: any problems?



# Dekker's solution

- Combined shared variables of Algorithms 1 and 2
- Process Pi

```
while (true) {
   flag[i] = true;
   while (flag[j]) {
      if (turn == j) {
         flag[i] = false;
         while (turn == j) ;   // wait
         flag[i] = true;
      }
   }
         /* critical section */
   turn = j;
   flag[i] = false;
         /* remainder section */
}
```

- Be polite: meet all three requirements; solve the critical-section problem for ***two*** processes

# Peterson's solution

- A simpler solution
  - combined shared variables of Algorithms 1 and 2
- Process $P_i$

```
do {
        flag [i]:= true;
        turn = j;
        while (flag [j] and turn == j) ;   // wait
                /* critical section */
        flag [i] = false;
                /* remainder section */
} while (1);
```

- Meet all three requirements; solve the critical-section problem for ***two*** processes

# This lecture

- Process synchronization
  - the producer-consumer problem
  - software solution for 2 processes
    - Peterson’s solution
- Explore further
  - Lamport’s bakery algorithm
    - for ***n*** processes
    - it’s time to google!

# Next lecture

- Process synchronization
  - other alternatives (read OSC7Ch6)