Part 4: Process Synchronization

- Background
- The Critical-Section Problem
- Synchronization Hardware
- Semaphores
- Classical Problems of Synchronization
- Monitor

* Important but difficult ☺
* Additional animations are in NTULearn.

Background

- Concurrent access to shared data may result in data inconsistency.
  - Inconsistent: the actual data value depends on the order of process executions (i.e., context switches can occur at an arbitrary code line)
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes.
- Suppose that we use a variable counter to keep track the number of items in the shared buffer in the producer-consumer example.

Bounded-Buffer

- Shared data

```
#define BUFFER_SIZE 10
typedef struct {
   . . .
} item;
item buffer[BUFFER_SIZE];
int in = 0;  //the next-to-fill empty slot
int out = 0;  //the next-to-process item
int counter = 0;
```

Bounded-Buffer (Cont.)

- Producer process

```
item nextProduced;
while (1) {

        while (counter == BUFFER_SIZE) ;

        buffer[in] = nextProduced;

        in = (in + 1) % BUFFER_SIZE;

        counter++;

}
```

Bounded-Buffer (Cont.)

- Consumer process

```
item nextConsumed;

while (1) {
        while (counter == 0) ; /* do nothing */
        nextConsumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        counter--;
}
```

Which parameter(s) are shared variables?

Bounded-Buffer (Cont.)

- The statements:
  - *counter*++;
  - *counter*--;

  may create inconsistent value due to *race condition*, i.e. the actual value depends on the order of processes that change this value. Hence the need of mechanism to synchronize processes to enforce consistency.

Race Condition

- counter++ could be implemented as

```
register1 = counter
register1 = register1 + 1   ←
counter = register1           Context switch
```

- counter-- could be implemented as

```
register2 = counter
register2 = register2 - 1   ←
counter = register2           Context switch
```

- Consider this execution interleaving with "counter = 5" initially:

S0: producer execute register1 = counter {register1 = 5}
S1: producer execute register1 = register1 + 1 {register1 = 6}
S2: consumer execute register2 = counter {register2 = 5}
S3: consumer execute register2 = register2 - 1 {register2 = 4}
S4: producer execute counter = register1 {counter = 6 }
S5: consumer execute counter = register2 {counter = 4}

The Critical-Section Problem

- n processes all competing to use some shared data
- Each process has a code segment, called *critical section*, in which the shared data is accessed.
  - Process may be changing shared variables, updating a shared table, writing a shared file, etc
  - At least one process modifies the shared data.
- Problem – to design protocol to ensure that when one process is executing in its critical section, no other process is allowed to execute in its critical section.

The Critical-Section Problem (cont.)

- Each process P_i must ask permission to enter critical section in entry section, may follow critical section with exit section, then remainder section

```
while (1) {
    entry section

            critical section

    exit section

            remainder section

}
```

Solution to Critical-Section Problem

Assumptions:

- Each process executes at a nonzero speed.
- No assumption about relative speed of the *n* processes.

1. **Mutual Exclusion**.
   If process P_i is executing in its critical section, then no other processes can be executing in their critical sections.

Solution to Critical-Section Problem (Cont.)

2. **Progress**.

If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely.

- If no process is executing in the critical section, then one process will be eventually selected.

3. **Bounded Waiting**.

A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

- A process can eventually get the chance to enter critical section.

Initial Attempts to Solve Problem

- Let's start with only 2 processes, P_0 and P_1
- General structure of process P_i (other process P_j)

```
while (1){
    entry section
        critical sections
    exit section
    remainder section
}
```

Software approaches without operating system support.

- Although P0 and P1 have the same structure, they may have different implementations/code on the four sections.
- Processes may share some common variables to synchronize their actions.

Algorithm 1

- Shared variables:

  int *turn*; /*initially *turn* = 0 */

  turn = *i* $\Rightarrow P_i$ can enter its critical section

Algorithm 1 (Cont.)

- Process P_i

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

Process P0

```
while (1){
    while (turn != 0) ;
        critical section
    turn = 1;
        remainder section
}
```

Process P1

```
while (1){
    while (turn != 1) ;
        critical section
    turn = 0;
        remainder section
}
```

Best practices for BMP are in the comment box.

Algorithm 1 (Cont.)

- Satisfies mutual exclusion.
- Do not satisfy progress.
  - For example, if turn=0 and P_1 again wants to enter the Critical Section, but P_1 cannot do so because P_0 is still in its Remainder Section.

Algorithm 2

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

Algorithm 2 (Cont.)

- Process P_i

```
while (1) {
    flag[i] = true;
    while (flag[j]) ;

        critical section

    flag [i] = false;

        remainder section

}
```

Algorithm 2 (Cont.)

- Satisfies mutual exclusion.
- Do not satisfy progress.
  - Consider the following execution sequence:

    T_0 : P_0 sets *flag*[0] = *true*

    T_1 : P_1 sets *flag*[1] = *true*

    Now P_0 and P_1 are looping forever in their respective **while** statements.

Algorithm 3

- Combined shared variables of algorithms 1 and 2.
- Process P_i

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

Algorithm 3 (Cont.)

- Meets all three requirements; solves the critical-section problem for two processes.

  (read p. 195-196 of text for the proof)

OS Support for Critical Section

- Previous software approaches are difficult to implement and inefficient.
  - They are for two processes only. What about multiple processes?
  - Solution: OS support.
- OS has the following three kinds of support for critical section (from low level to high level)
  - Synchronization hardware
  - Semaphore
  - Monitor

Synchronization Hardware

- Modern machines provide special atomic hardware instructions
  - Atomic = non-interruptable
- *TestAndSet*: Test and modify the content of a word atomically.

```
boolean TestAndSet (boolean *target) {
        boolean rv = *target;
        *target = true;
        return rv;
}
```

$\rightarrow$ Allow context switches

No context switches

$\rightarrow$ Allow context switches

Mutual Exclusion with Test-and-Set

- Shared data: boolean *lock* = *false;*
- Process P_i

```
while (1) {
        while(TestAndSet(&lock));        Acquire lock
            critical section
        lock = false;                    Release lock
            remainder section
}
```

Semaphore

- Semaphore ***S*** – integer shared variable
- **Only** accessed by two **atomic** operations : *wait* and *signal*
  - *Wait (S):* if S>0, S--; otherwise, wait.
  - *Signal (S):* S++.

Enter waiting state, or busy loop waiting

- Two types of semaphore
  - *Counting* semaphore: integer value can range over an unrestricted domain.
  - *Binary* semaphore: integer value can range only between 0 and 1

Example: Critical Section of *n* Processes

- Shared variables
  - *semaphore mutex;*
  - initially *mutex* = 1
- Process P_i

```
while(1) {
    wait(mutex);
        critical section
    signal(mutex);
        remainder section
};
```

Classical Semaphore Implementation

- The classical definitions of *wait* and *signal* that uses **busy waiting**:

```
wait (S):
        while (S<= 0);
        S--;

signal (S): S++;
```

- Pros: no context switches for short critical sections (good for a short waiting time).
- Cons: inefficient if the critical section is long.

Current Semaphore Implementation: Blocking

- Define a semaphore as a record

```
typedef struct {
    int value;
    struct process *L;
} semaphore;
```

L is a queue that stores the processes waiting to enter the critical section (in the form of PCB)

- Need two simple operations:
  - *block*( ): blocks the current process.
  - *wakeup*( ): resumes the execution of a blocked process in S.L.

Semaphore Implementation (Cont.)

```
block( ):
    dequeue current process from ready queue;
    enqueue current process to S.L;

wakeup( ):
    dequeue a process P from S.L;
    enqueue process P to ready queue;
```

Semaphore Implementation (Cont.)

- Busy waiting wastes CPU time. Hence redefine *wait* and *signal* operations.

wait(*S*):

```
        S.value--;
        if (S.value < 0) {
                block( );
        }
```

If S.value=-2, how many processes are there waiting in S.L?

Semaphore Implementation (Cont.)

```
signal(S):
        S.value++;
        if (S.value <= 0) {
            wakeup(P);
        }
```

Semaphore as General Synchronization Tool

- Execute *B* in P_j only after *A* executed in P_i
- Use semaphore *flag* initialized to **0**
- Code:

P_i	P_j
⋮	⋮
A	*wait*(*flag*)
signal(*flag*)	*B*

Deadlock

- Deadlock – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes.
- Let *S* and *Q* be two semaphores initialized to 1

P_0	P_1
wait(*S*);	*wait*(*Q*);
wait(*Q*);	*wait*(*S*);
⋮	⋮
signal(*S*);	*signal*(*Q*);
signal(*Q*);	*signal*(*S*);

Starvation

- Starvation - indefinite postponement.
  - A process may never have a chance to be removed from the semaphore queue (i.e. S.L) due to queue discipline. Examples are:
    - ❄ *Priority-based: low priority process may starve if higher priority processes keep joining the queue*
    - ❄ *LIFO: late arrival processes will take advantage but earlier arrival processes may starve.*

Classical Problems of Synchronization

- Bounded-Buffer Problem
- Dining-Philosophers Problem
- Readers and Writers Problem

Best Practices for Semaphores

- **Step 1:** understand the application scenario and identify the following:
  - Shared variables,
  - Shared resources.
- **Step 2:** protect the shared variables:
  - Identify the critical section,
  - Use binary semaphore, and add entry section (wait) and exit section (signal).
- **Step 3:** protect the shared resources:
  - Identity each kind of resources; one semaphore for one **kind**.
  - Initial value: the number of resource instances.
  - When the resource is requested/consumed: wait.
  - When the resource is released: signal.

Bounded-Buffer Problem

- Consider multiple producers/consumers
- Shared data and resources (Step 1)

```
#define BUFFER_SIZE n
Typedef struct {
      . . .
} item;
item buffer[BUFFER_SIZE];
semaphore full, empty, mutex;
full =0; empty = n; mutex =1;
```

Binary semaphore for mutual exclusion on shared variables.

Counting semaphore for multiple instances of resources.

Bounded-Buffer Problem (Cont.)

- Producer process (Step 2)

No synchronization:

```
item nextProduced;
while (1) {
        …
    produce nextProduced;
        …
    add nextProduced to buffer;
        …
}
```

```
item nextProduced;
while (1) {
        …
    produce nextProduced;
        …
    wait(mutex);
        …
    add nextProduced to buffer;
        …
    signal(mutex);

}
```

Bounded-Buffer Problem (Cont.)

- Producer process (Step 3)

```
item nextProduced;
while (1) {
      ...
   produce nextProduced;
       ...
   wait(mutex);
      ...
   add nextProduced to buffer;
      ...
   signal(mutex);

}
```

```
item nextProduced;
while (1) {
      ...
   produce nextProduced;
       ...        Consume one empty slot.
   wait(empty);
   wait(mutex);
      ...
   add nextProduced to buffer;
      ...
   signal(mutex);
   signal(full);
}          Generate one item.
```

Bounded-Buffer Problem (Cont.)

- Consumer process

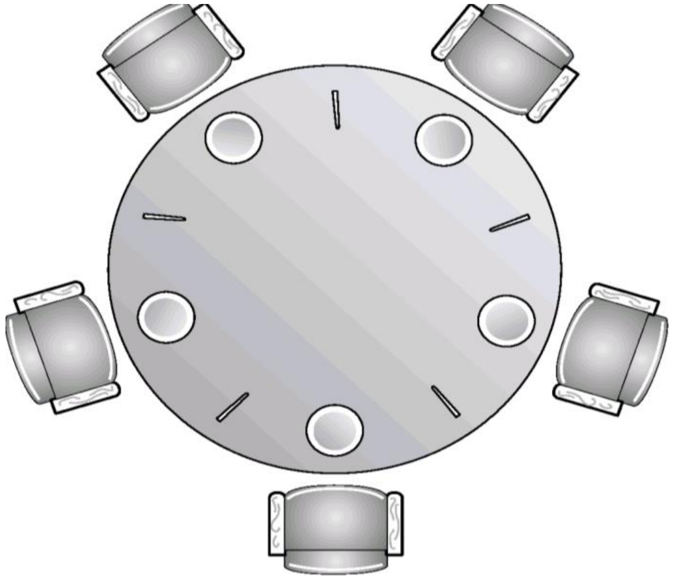
```
item nextConsumed;
while (1) {
    wait(full);
    wait(mutex);
        …
     nextConsumed = an item from buffer;
        …
    signal(mutex);
    signal(empty);
        …
    consume the item nextConsumed;
        …
}
```

Example: Dining-Philosophers

Question: One simple solution to dining philosophers problem is to represent each chopstick by a semaphore:

semaphore *chopstick[5];*

where all the elements of chopstick are initialized to 1.



Example: Dining-Philosophers (Cont.)

A philosopher tries to grab the chopsticks by execution the operation *wait* on its and its neighbor's semaphores, and the chopsticks are released by execution the operation *signal* on the appropriate semaphores. Thus, the code for philosopher *i* is as follows:

Example: Dining-Philosophers (Cont.)

- Philosopher *i*:

```
while (1) {
  wait(chopstick[i]);
  wait(chopstick[(i+1)%5]);
     …
    eat
     …
  signal(chopstick[i]);
  signal(chopstick[(i+1)%5]);
     …
    think
     …
  }
```

Example: Dining-Philosophers (Cont.)

a) Discuss the problem with the above solution;

b) Suggest remedies to the problem.

Example: Dining-Philosophers (Cont.)

Solution:

a) Suppose that all five philosophers become hungry simultaneously, and each grabs his left chopstick. All the <u>semaphores</u> of chopstick will now be equal to 0. When each philosopher tries to grab his right chopstick, he will be delayed forever.→ Deadlock

Example: Dining-Philosophers (Cont.)

b) There are several possible remedies to the above problem:

- Allow at most four philosophers to be hungry simultaneously.
- Allow a philosopher to pick up his chopsticks only if both chopsticks are available.
- Use an asymmetric solution, that is, an odd philosopher picks up first his left chopstick and then his right chopstick, whereas an even philosopher picks up his right chopstick and then his left chopstick.

Readers-Writers Problem

- Scenario on a database:
  - A writer requires exclusive accesses.
  - Multiple readers can concurrently accesses.
    - ❄ *The first reader: before reading, it blocks the writers.*
    - ❄ *The last reader: after reading, it allows writers*
- Shared data

```
semaphore mutex=1, wrt=1;
int readcount =0;
```

The database

Shared variable (#readers)

- Writer process

```
wait(wrt);
     ...
  writing is performed
     ...
signal(wrt);
```

Readers-Writers Problem (Cont.)

- Reader process

When a reader comes.

```
wait(mutex);
readcount ++;
if (readcount == 1) wait(wrt);
signal(mutex);
```

The first reader? Request the database

```
    …
  reading is performed
    …
```

When a reader leaves.

```
wait(mutex);
readcount --;
if (readcount == 0) signal(wrt);
signal(mutex):
```

The last reader? Release the database

Monitor

- High-level synchronization construct that allows the sharing of variables among concurrent processes.
- Monitor is a collection of procedures and data.
- Processes may call procedures within monitor to access serially reusable shared resources.
- Only one process can be active in the monitor at any one time (i.e., only one process can be executing a monitor procedure at any one time).
- Data within monitor can only be accessed by procedures within it.

This slide is not examinable.