Process Synchronization (part 2)

ECE595, Jan 23

Y. Charlie Hu

Review: Process synchronization

- Cooperating processes need to
  - share data
  - synchronize access to shared data
- Accessing shared data needs to be in CS
- Other types of synchronization more complex
- Synchronization without OS help is hard
- Sync primitives supported by OS
  - Lock() is simple, but not powerful enough
  - More powerful ones were invented
    - Semaphore
    - Condition variables

Semaphore

- A synchronization variable that takes on non-negative integer values
  - Invented by Edsger Dijikstra in the mid 60's
- Two primitve operations
  - wait(semaphore): an <u>atomic</u> operation that waits for semaphore to become greater than 0, then decrements it by 1
  - signal(semaphore): an <u>atomic</u> operation that increments semaphore by 1

Semaphore

```
wait(S) {
   while (S<=0);
   S--;
}

signal(S) {
   S++;
}
```

- In reality, wait(S) is not implemented as above!
- Semaphores aren't provided by hardware (why not?) – we'll discuss OS implementations next time

Binary Semaphore

Init: S =1;

```
wait(S) {                    signal(S) {
   while (S==0);                if (S == 0) S++;
   S--;                      }
}
```

- Binary semaphores: only take 0 or 1
- Sounds familiar?
  - S=0 → someone is holding the lock!

semaphores vs. locks: fundamental difference?

Semaphores

```
wait(S) {                    signal(S) {
   while (S<=0);                S++;
   S--;                      }
}
```

Binary Semaphore (lock)

```
wait(S) {                    signal(S) {
   while (S==0);                if (S == 0) S++;
   S--;                      }
}
```


semaphore has built-in counting!

- signal(S) simply increments S
  - "just produced an item"
  - S value == how many items have been produced
- wait(S) will return without waiting only if S > 0;
  - Wait(S) is saying "waited until there is at least one item, and then just consumed an item"

Two usages of semaphores

- For mutual exclusion:
  - to ensure that only one process is accessing shared info at a time.
  - Semaphores or binary semaphores?
- For condition synchronization:
  - to permit processes to wait for certain things to happen
  - Semaphores or binary semaphores?

Producer & Consumer (1-pool version)

- Define constraints (what is “correct”)
  - Consumer must wait for producer to fill buffer (mutual excl. or condition sync?)
  - Producer must wait for consumer to empty buffer, if all buffer space is in use (mutual excl. or condition sync?)
  - Only one process must manipulate buffer at once (mutual excl. or condition sync?)
- Use a separate semaphore for each constraint
  - Full = 0
  - Empty = N
  - Mutex = 1

Producer & Consumer – solution using locks?

Producer	Consumer
while (1) {	While (1) {
produce an item;	while (buffer is empty);
while (buffer is full);	remove an item;
insert item into buffer	consume the item
}	}

Deep thinking

- Why does producer wait(EMPTY) but signal(FULL)
  - Explain in terms of creating and destroying resources
- Is the order of signal()’s important?
- Is the order of wait()’s important?
- How would this be extended to have > 1 consumers?

Break

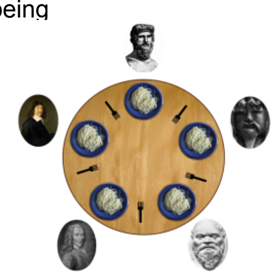
Classic Synchronization Problems

1. Producer-consumer problem (bounded buffer problem)
2. Readers-writers problem
3. Dining philosophers problem

Dining philosophers problem

Abstraction of concurrency-control problems

The need to allocate several resources among several processes while being deadlock-free and starvation-free



Classic Synchronization Problems

1. Producer-consumer problem (bounded buffer problem)
2. Readers-writers problem
3. Dining philosophers problem

Readers-Writers problem

Abstraction of concurrent access to shared data problem

- A data object is shared among multiple processes

Reader:

```
While (1) {

acq(mutex)
read();
rel(mutex)

}
```

Writer:

```
While (1){

acq(mutex)
write();          /* abstraction */
rel(mutex)

}
```

Readers-Writers problem

Abstraction of concurrent access problem

- A data object is shared among multiple processes
- Allow concurrent reads, but exclusive writes
  - Implication: need to move read() and write() outside Critical Sec
  - Can we do it using local flags?
  - Can we use semaphore to count readers/writers?

Reader:	Writer:
acq(mutex) ???? rel(mutex)	acq(mutex) ???? rel(mutex)
read();	write();
acq(mutex) ??? rel(mutex)	acq(mutex) ??? rel(mutex)

Readers-Writers problem

Abstraction of concurrent access problem

- A data object is shared among multiple processes
- Allow concurrent reads, but exclusive writes
- Solution needs lock, counting, and semaphores!
- Constraints:
  - Writers can only proceed if there are no active readers/writers
    ➔ use semaphore OKtoWrite
  - Readers can proceed only if there are no active/waiting writers
    ➔ use semaphore OKtoRead
  - To keep track of how many are reading / writing / waiting
    ➔ use some shared variables, called *state variables*
  - Only one process manipulates state variable at once
    ➔ use a lock Mutex

Readers-Writers problem (cont)

- State variables:
  - AR = number of active readers
  - WR = number of waiting readers
  - AW = number of active writers
  - WW = number of waiting writers

  AW is always 0 or 1

  AR and AW can not both be non-zero
- Initialization:
  - OKtoRead = 0;
  - OKtoWrite = 0;
  - Mutex = 1;
  - AR = WR = AW = WW = 0;
- Scheduling: writers get preference

Readers-Writers problem (cont)

- Reader

```
acq(mutex)                                        acq(mutex);



                                                  rel(mutex);

rel(mutex)
wait(OKtoRead);
read necessary data;
```

Readers-Writers problem (cont)

- Writer

```
acq(mutex)





rel(mutex)
wait(OKtoWrite);
write necessary data;
```

```
acq(mutex);




rel(mutex);
```


Readers-Writers problem (cont)

- Reader

```
acq(mutex)
if (????) {
  signal(OKtoRead);
  AR ++;
} else {
  WR ++;
}
rel(mutex)
wait(OKtoRead);
read necessary data;
```

```
acq(mutex);
AR--;
if (????) {
  V(OKtoWrite);
  AW ++;
  WW --;
}
rel(mutex);
```


Readers-Writers problem (cont)

- Writer

```
acq(mutex)
if (????) {
  signal(OKtoWrite);
  AW ++;
} else {
  WW++;
}
rel(mutex);
wait(OKtoWrite);
write necessary data;
```

```
acq(mutex);
AW --;
if (????) {
  signal(OKtoWrite);
  AW ++;
  WW --;
} else while (????) {
  signal(OKtoRead);
  AR ++;
  WR --;
}
rel(mutex);
```


What happens if

- Reader enters and leaves system
- Write enters and leaves system
- Two writers enter system
- Two readers (a,b) enter system
- Writer(c) enters system and waits
- Reader(d) enters system and waits
- Readers(a,b) leave system, write(c) continues
- Write(c) leaves system, last reader(d) continues and leaves

Questions:

- In case of conflict between readers and writers, who gets priority?
  - Readers can get locked out
- Is the WW necessary in the writer's first if?
  - No: if there is a waiting writer, there must be an active writer or at least one active reader
- Can OKtoRead ever get greater than 1? What about OKtoWrite?
  - Yes, no
- Is the first writer to execute P(mutex) guaranteed to be the first writer to access the data
  - No, waiting writers can get granted in any order

Reading Assignment

- Chapter 6