Operating system concepts

Process Synchronization (Producer-consumer, critical section, mutex)

Slides Set #10

By Prof K R Chowdhary

JNV University

2023

Interprocess Communication

- A concurrent process may be either *independent* processes or *cooperating* processes.
- Reasons for providing process cooperation:
  - Information sharing.
  - Computation speedup.
  - Modularity.
  - Convenience.
- Cooperating processes require an interprocess communication (IPC) mechanism to exchange data and information.
- Two fundamental models of interprocess communication: *shared memory* and *message passing*.

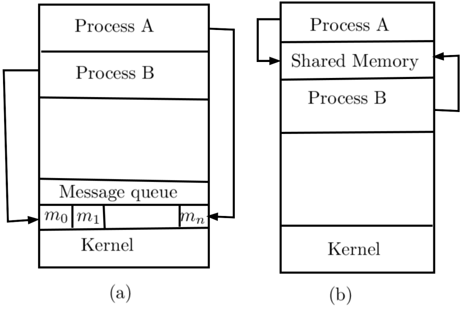

Figure 1: (a) Message passing, (b) Shared Memory

1. Shared-Memory System, 2. Message passing

- **Shared Memory**:
  - Interprocess communication using *shared memory* requires communicating processes to establish a region of shared memory (see Fig. 1).
  - The form of the data and the location are determined by these processes and are not under the operating system's control.
  - Ensure that they are not writing to the same location *simultaneously*.
- **Message Passing**:
  - $send(A, message)$:Send a message to mail box A.
  - $receive(A, message)$:Receive a message from mailbox A
  - Sockets: For network communications

Producer-consumer Problem

- Processes can execute *concurrently* or in *parallel*.
- The concurrent or parallel execution can contribute to issues involving the *integrity* of data shared by several processes.
- Consider the bounded buffer (buffer size fixed). This allows for at most "BUFFERSIZE - 1" items in the buffer.

-:Producer Process code:-

```
while (true){
  /* produce an item in next produced */
  while (counter == BUFFERSIZE);
     /* do nothing */
  buffer[in] = next_produced;
  in = (in + 1) % BUFFERSIZE;
  counter++;
}
```

Producer-consumer problem...

-:Consumer process code:-

```
while (true) {
  while (counter == 0);
     /* do nothing */
  next_consumed = buffer[out];
  out = (out + 1) % BUFFERSIZE;
  counter--;
  /* consume the item in next consumed */
}
```

We would arrive at this incorrect state because we allowed both processes to manipulate the variable counter concurrently. (race!)

- Questions: On producer-consumer problem.
  - Meaning of "while (counter == BUFFERSIZE);" in producer?
  - Is buffer[] global array?
  - Are "in" and "out" global variables?
  - Meaning of "while (counter == 0);" in consumer?

Critical-Section Problem

- Consider a system consisting of *n* processes $\{P_0, P_1, ..., P_{n-1}\}$.
- Each process has a segment of code, called a *critical section*, in which the process may be changing common variables,..
- Each process must request permission to enter its critical section.
- The critical section may be followed by an exit section.

```
do {
    --- entry into section ---
         [critical section]
    --- exit from section
        remainder code
}  while true;
```

- Questions:
  - Give any five examples, where in the operating the producer-problem occurs?
  - What is meaning of entry into critical section?

Critical-Section Problem...

- A solution to the critical-section problem must satisfy these requirements:
  1. Mutual exclusion.
  2. Progress (selection of which goes into critical section cannot be postponed indefinitely).
  3. Bounded waiting (for critical section).
- Questions:

- What operation happens in the critical section?
- Examples of critical section in real-life?
- What is meaning of mutual-exclusion?
- Progress means what?
- Difference between point 2 and 3 above?

Handling Critical-Section in **Kernel processes**

- Two general approaches are used to handle critical sections in operating systems: *preemptive kernels* and *nonpreemptive kernels*.
- Obviously, a nonpreemptive kernel is essentially free from race conditions on *kernel data structures*
- a preemptive kernel is more suitable for real-time programming,
- Peterson's Solution (algorithm) for handling critical section: SW solution

```
//whose turn it is to enter criti.sec. (1 -> P1, 2->P2)
int turn;
//flag[0] =true; -> P0 is ready to enter critical section
boolean flag[2];
```

Handling Critical-Section in Kernel processes...

- Peterson's solution requires the two processes to share two data items:

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

- The variable *turn* indicates whose turn it is to enter its critical section. That is, if turn == i, then process Pi is allowed to execute in its critical section.
- Question:
  - What two data items are shared between two processes?
  - How it is ensured by above code that only one process enters the critical section?

Handling Critical-Section in Kernel processes...

- We now prove that this solution is correct. We need to show that:
  1. Mutual exclusion is preserved.
  2. The progress requirement is satisfied.
  3. The bounded-waiting requirement is met.
- To show properties 2 and 3 above, note that a process Pi can be prevented from entering the critical section only if it is stuck in the while loop with the condition flag[j] == true && turn == j;

Mutex Locks

- The simplest of these tools is the mutex lock. (In fact, the term mutex is short for mutual exclusion.)
- A mutex lock has a boolean variable *available* whose value indicates if the lock is available or not. If the lock is available, a call to acquire() succeeds, and the lock is then considered unavailable.
- The definition of acquire() is as follows:

```
acquire() {
    while (!available);
       /* busy wait */
    available = false;
}
do {
   acquire_lock()
      critical section
   release_lock()
      remainder section
} while (true);
```

The definition of release() is as follows:

```
release() {
   available = true;
}
```

Mutex Locks...

- Questions:
  - What are the disadvantages of mutex lock (called also spinlock)?
- What is meaning of "spinlock"?
  - "Busy waiting wastes CPU cycles" means what?
  - Are there possible advantages of spinlocks?
  - Does mutex prevent the race condition?

Building a mutex Lock

- Goals of a lock implementation:
  - *Mutual exclusion* (obviously!)
  - *Fairness*: all threads should eventually get the lock, and no thread should starve
  - *Low overhead*: acquiring, releasing, and waiting for lock should not consume too many resources
- Implementation of locks are needed for both user-space programs (e.g., pthreads library) and kernel code
- Implementing locks needs support from hardware and OS
- Questions:
  - What are goals of implementation of mutex lock?
  - What are functions of “available”, “acquire” and “release”?

Critical section and locks

- Consider update of shared variable *balance* in C code with operation:

  ```
  balance = balance + 1;
  ```

- We can use a special lock variable to protect it

  ```
  lock_t mutex; //some globally allocated lock 'mutex'
  ....
  lock(&mutex);
  balance = balance +1;
  unlock(&mutex);
  ```

- All threads accessing a critical section share a lock (function())
- Only one threads succeeds in locking, i.e., owner of lock
- Other threads that try to lock cannot proceed further until lock is released by the *owner*
- *pthreads* library in Linux provides such locks

Is disabling interrupts enough?

- Is this enough?

```
void lock(){
    DisableInterrupts();
}
void unlock(){
    EnableInterrupts();
}
```

- No, not always!
  - Many issues here:
  - Disabling interrupts is a privileged instruction and program can misuse it (e.g., run forever)
  - Will not work on multiprocessor systems, since another thread on another core can enter critical section
- This technique is used to implement locks on single processor systems inside the OS
  - Need better solution for other situations