ICS431 Ch. 6 & 7

Alfaifi, Ammar

Chapter 1

Synchronization Tools

1.1 Background

- A **cooperating process** is one that can affect or be affected by other processes executing in the system.
- Cooperating processes can either directly share a logical address space or be allowed to share data through message passing.
- Concurrent access to shared data may result in data inconsistency, so synchronization is a must!
- Recall the **unbound buffer** places no practical limit on the size of the buffer. The consumer may have to wait for new items, but the producer can always add new items.
- For the **bounded buffer**, it assumes a fixed buffer size. The consumer must wait if the buffer is empty, and the producer must wait if it's full.
- We consider the code of bounded buffer from Section 3.5, we add a new variable `count`, to track the number of items in the buffer.
- As you see in the consumer's and producer's codes, page 274, the producer increment `count` and the producer tries to decrement it each time. When running concurrently, there are clashing!
- When several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particulate order in which the access takes place is called **race condition**.
- Since we are interleaving the lower-level statements of both codes (but the order withing each high-level is preserved), both `count++` and `count--` are executed using the same old value, resulting wrong result; this is a race condition.

1.2 The Critical-Section Problem

- Consider a system with n processes $P_0, P_1, \dots, P_{n-1}$. Each process ha a segment of code is a **critical section**, meaning a process can modify a data that's shared with at least one other process.
- The section of code implementing this request is called **entry section**. There can be also **exit section**.
- A solution to he critical-section problem must satisfy the following
  1. **Mutual exclusion** if process P_i is executing it its critical section, no other processes can be executing in their critical sections.
  2. **Progress** if no process is executing in its critical section and some processes want to enter their critical sections, then only those aren't executing in their remainder section can decide which will enter its critical section next.
  3. **Bounded waiting** a bound (limit) on the number of times that other processes are allowed to enter their critical section after a process made a request to enter its critical section and that request.
- Let `next_available_pid` be the pid to be assigned to a process after calling `fork()`. If `fork()` is called twice these will lead to a race condition in a kernel data structure.
- This problem can be avoided in single-core environment, by disabling interrupts while a modifying a shared variable. But this inefficiency solution, and not feasible in multiprocessor environment.
- Two approaches are used to handle this problem:
  1. **Preemptive kernels** allows a process to be preempted while it is running in kernel mode.
  2. **Nonpreemptive kernels** does not allow a process running in kernel mode to be preempted; a kernel-mode process will run until it exits kernel mode, blocks, or yields control of CPU back.
- Obviously, nonpreemptive doesn't have the race condition problem on kernel data structure, since only one process at time can run.

1.3 Peterson's Solution

- This is a software-based solution to the critical-section problem.
- It requires the two processes to share two data items as, written in `C++`

```
int turn;
boolean flags[2];
```

- `turn` indicates whose turn it is to enter its critical section. if `turn == i`, then process P_i is allowed to execute its critical section.

- `flag` array indicates if a process is ready to enter its critical section; if `flag[i]` is `true`, P_i is ready to enter its critical section.
- The process P_i code should be

```
While (true) {
flag[i] = true;
turn = j;
while (flag[j] && turn == j)
;

/* Here is the critical section */

flag[i] = false;

/* remainder section */
}
```