CS370 Operating Systems

Colorado State University
Yashwant K Malaiya

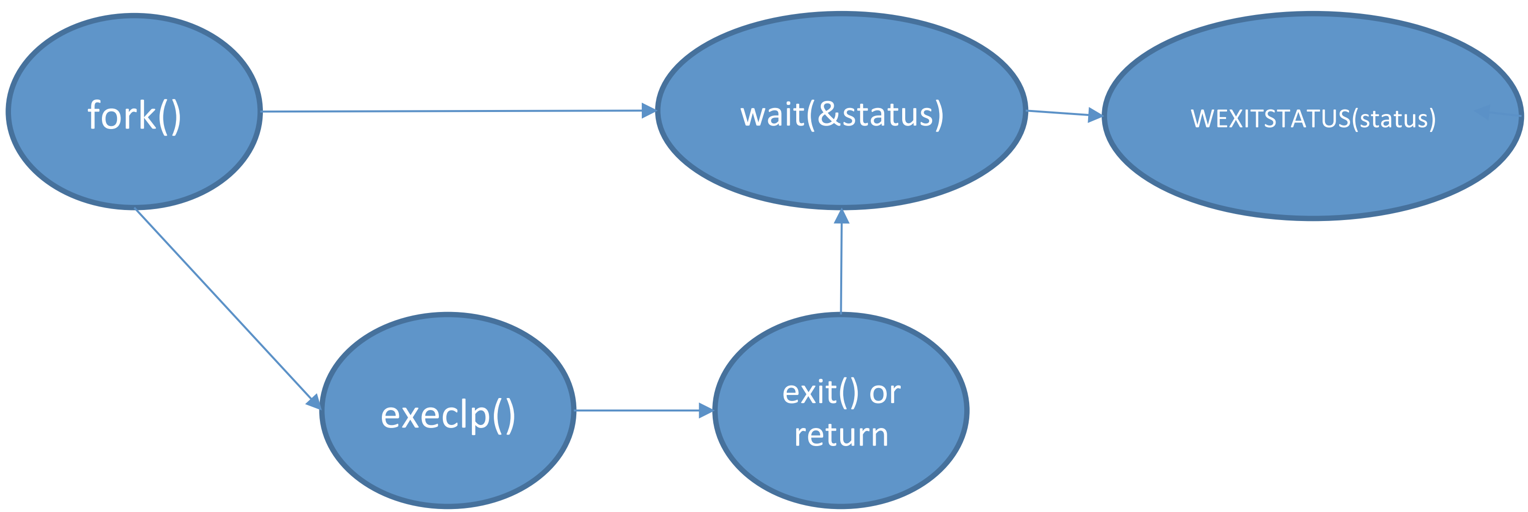
Fall 2016 Lecture 16

Slides based on

- Text by Silberschatz, Galvin, Gagne
- Various sources

Questions from Last Time

- Using fork(), exec( ), wait( )



Questions from Last Time

- Fork( ) returns 0 in child process, and child PID in parent. How does it return in two places?
- Why do determinist modeling?
- Simulation for evaluating an algorithm? validity?
- Little's formula: in steady state,
  - average queue length =av arrival rate x av waiting time in queue
- Critical section: why needed? How to implement? More coming up
- Round robin for multi-level queue- why?
- Multi-level: what after low priority processes are done?
- Scheduling algorithm in OSX? Multilevel feedback queue

Process Synchronization: Outline

- Process synchronization: critical-section problem to ensure the consistency of shared data
- Software and hardware solutions of the critical-section problem
  - Peterson's solution
  - Atomic instructions
  - Mutex locks and semaphores
- Classical process-synchronization problems
- Bounded buffer, Readers Writers, Dining Philosophers
- Another approach: Monitors

- We saw race condition between counter ++ and counter --

Too Much Milk Example

	Person A	Person B
12:30	Look in fridge. Out of milk.	
12:35	Leave for store.	
12:40	Arrive at store.	Look in fridge. Out of milk.
12:45	Buy milk.	Leave for store.
12:50	Arrive home, put milk away.	Arrive at store.
12:55		Buy milk.
1:00		Arrive home, put milk away. Oh no!

Too Much Milk Example

	Person A	Person B
12:30	Look in fridge. Out of milk.	
12:35	Leave for store.	
12:40	Arrive at store.	Look in fridge. Out of milk.
12:45	Buy milk.	Leave for store.
12:50	Arrive home, put milk away.	Arrive at store.
12:55		Buy milk.
1:00		Arrive home, put milk away. Oh no!

Critical Section

Solution to the "*race condition*" problem: critical section

- Consider system of ***n*** processes $\{p_0, p_1, \ldots p_{n-1}\}$
- Each process has **critical section** segment of code
  - Process may be changing common variables, updating table, writing file, etc
  - When one process in critical section, no other may be in its critical section
- ***Critical section problem*** is to design protocol to solve this
- Each process must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder section**

Race condition: when outcome depends on timing/order that is not predictable

Critical Section

```
do {

    entry section

        critical section

    exit section

        remainder section

} while (true);
```

Request permission to enter

Housekeeping to let processes to enter other

Solution to Critical-Section Problem

We want

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

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

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

- Speed assumptions
  - Assume that each process executes at a nonzero speed
  - No assumption concerning **relative speed** of the ***n*** processes

Critical-Section Handling in OS

Two approaches depending on if kernel is preemptive or non- preemptive

- **Preemptive** – allows preemption of process when running in kernel mode. Must ensure shared kernel data is free from race conditions

- **Non-preemptive** – runs until exits kernel mode, blocks, or voluntarily yields CPU
  - Essentially free of race conditions in kernel mode

Peterson’s Solution

- **Software solution** to the critical section problem
  - Restricted to two processes
- No guarantees on modern architectures

  Machine language instructions such as load and store implemented differently
- Good algorithmic description
  - Can shows how to address the 3 requirements

Peterson's Solution

- Assume that the `load` and `store` machine-language instructions are atomic; that is, cannot be interrupted
- The two processes P_i and P_j share two variables:
  - `int turn;`
  - `Boolean flag[2]`
- The variable `turn` indicates whose turn it is to enter the critical section
- The `flag` array is used to indicate if a process is ready to enter the critical section. `flag[i] = true` implies that process P_i is ready!

Algorithm for Process P_i

For process Pi

Being nice!

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

- The variable `turn` indicates whose turn it is to enter the critical section
- `flag[i] = true` implies that process P_i is ready!

Colorado State University

Peterson's solution: Mutual exclusion

```
while (flag[j] && turn = = j);
```

- P_i enters critical section only if flag[j] == false OR turn == i
- If both processes enter critical section at the same time
  - flag[0] == flag[1] == true
  - **But** turn can be 0 or1, not BOTH
- f P_j entered critical section
  - flag[j] == true AND turn == j
  - persist as long as Pj is in the critical section

Peterson's : Progress and Bounded wait

- P_i can be stuck only if flag[j]=true AND turn==j
  - If P_j is *not ready*: flag[j]== false, and P_i can enter
  - Once P_j *exits*: it resets flag[j] to false
- If Pj resets flag[j] to true
  - Must set turn = i;
- Pi **will enter** critical section (*progress*) after at most one entry by Pj (*bounded wait*)

Note: there exists a generalization of Peterson's solution for more than 2 processes, but bounded waiting is not assured.

Synchronization: Hardware Support

- Many systems provide hardware support for implementing the critical section code.
- All solutions below based on idea of **locking**
  - Protecting critical regions via locks
- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
    - Operating systems using this not broadly scalable
- Modern machines provide special atomic hardware instructions
    - **Atomic** = non-interruptible
  - test memory word and set value
  - swap contents of two memory words

Solution to Critical-section Problem Using Locks

```
do {
     acquire lock
               critical section
     release lock
               remainder section
} while (TRUE);
```

Hardware approaches

1. test memory word and set value
2. swap contents of two memory words

test_and_set Instruction

Definition:

```
boolean test_and_set (boolean *target)
{
     boolean rv = *target;
     *target = TRUE;
     return rv:
}
```

Executed **atomically**

1. Returns the original value of passed parameter
2. Set the new value of passed parameter to "TRUE".

Solution using test_and_set()

- Shared Boolean variable lock, initialized to FALSE
- Solution:

```
do {
     while (test_and_set(&lock)) ; /* do nothing */

            /* critical section */

      …..
     lock = false;
            /* remainder section */
    …  ..

  } while (true);
```

To break out:
Return value of TestAndSet should be FALSE

Lock FALSE: not locked.
If two TestAndSet() are executed *simultaneously*, they will be executed *sequentially* in some arbitrary order

compare_and_swap Instruction

Definition:

```
int compare_and_swap(int *value, int expected, int new_value)
{
        int temp = *value;

        if (*value == expected)
           *value = new_value;
     return temp;
 }
```

Executed **atomically**

1. Returns the original value of passed parameter “value”
2. Set the variable “value” to “new_value” but only if “value” ==“expected”.

Set Lock to locked (1), but only if it is open (0)

Colorado State University

Solution using compare_and_swap

- Shared integer “lock” initialized to 0 0 = unlocked;
- Solution:

Expected=0, new=1

```
do {
  while (compare_and_swap(&lock, 0, 1) != 0)
   ; /* do nothing */
   /* critical section */
lock = 0;
   /* remainder section */
} while (true);
```

- **Does not guarantee bounded waiting.** But see next.

Bounded-waiting Mutual Exclusion with test_and_set

```
For process i:
do {
   waiting[i] = true;
   key = true;
   while (waiting[i] && key)
      key = test_and_set(&lock);
   waiting[i] = false;
   /* critical section */
   j = (i + 1) % n;
   while ((j != i) && !waiting[j])
      j = (j + 1) % n;
   if (j == i)
      lock = false;
   else
      waiting[j] = false;
   /* remainder section */
} while (true);
```

Shared Data structures initialized to FALSE

- **boolean waiting[n];**
- **boolean lock;**

The entry section for process i : First process to execute TestAndSet will find key == false ; ENTER critical section, EVERYONE else must wait

The exit section for process i:

Part I: Finding a suitable waiting process j, or make lock FALSE.

Bounded-waiting Mutual Exclusion with test_and_set

The previous algorithm satisfies the three requirements

- **Mutual Exclusion**: The first process to execute TestAndSet(lock) when lock is false, will set lock to true so no other process can enter the CS.
- **Progress**: When a process exits the CS, it either sets lock to false, or waiting[j] to false, allowing the next process to proceed.
- **Bounded Waiting**: When a process exits the CS, it examines all the other processes in the waiting array in a circular order. Any process waiting for CS will have to wait at most n-1 turns