CMSC 321: Operating Systems

Lecture 5

Concurrency Issues
What is Concurrency?

- Consider several processes/threads executing “simultaneously”

- Design issues:
  - communication among processes
  - competing for resources
  - sharing resources
  - synchronizing activities of processes
  - allocation of CPU time to processes (involves scheduling decisions)
IPC

- **InterProcess Communication (IPC):**
  - communication between/among processes

- **Three issues:**
  - how to pass information
  - ensuring that ≥ 2 processes don’t “snafu” when performing *critical activities*
  - ensuring proper sequencing when dependencies present

- **First issue:**
  - pipes, signals
  - alternatives: message passing (§2.2.8), shared memory

- **Focus on second issue**
Shared Memory, 1 CPU

```c
void echo()
{
    ch_in = getchar();
    ch_out = ch_in;
    putchar(ch_out);
}
```

- Single-processor, multiprogramming system
  - variables `ch_in`, `ch_out` are global
  - two processes, P1 and P2

- Consider:
  - P1 invokes `echo()`, executes first line, interrupted
  - P2 activated, invokes `echo()`, completes
  - P1 resumes

- What is the output?
void echo()
{
    ch_in = getchar();
    ch_out = ch_in;
    putchar(ch_out);
}

Process P1

ch_in = getchar();

ch_out = ch_in;

putchar(ch_out);

Process P2

.

ch_in = getchar();

ch_out = ch_in;

putchar(ch_out);
What’s the Problem?

• Shared access to common resource
  - the global variable

• How can we fix?
  - ensure only one process in `echo()`
  - P1 enters `echo()`, interrupted
  - P2 activated, invokes `echo()`, must block
  - P1 eventually resumes, completes
  - P2 later resumes, no block

• Lesson: *protect shared resources*
  - control the code that accesses the resource
void echo()
{
    ch_in = getchar();
    ch_out = ch_in;
    putchar(ch_out);
}

Process P1
.
ch_in = getchar();
.
ch_out = ch_in;
putchar(ch_out);
.
.

Process P2
.
ch_in = getchar();
.
ch_out = ch_in;
.
putchar(ch_out);
.
.
Race Condition

• Two or more processes/threads:
  – reading/writing shared data
  – final result depends on order of execution

• Previous example is a race condition
  – P1 and P2 “race” to write to ch_in

• Another example:
  – P3 and P4 share globals \( b, c \)
  – Initially, \( b = 1 \) and \( c = 2 \)
  – P3 executes \( b = b + c; \)
  – P4 executes \( c = b + c; \)
Race Condition

- Another example: printing
  - items appended to a spool buffer
The General Problem

• Two or more processes need access to a shared resource

• Each process unaware of the other, should be unaffected by the other

• No exchange of info between processes

• If both want the resource:
  – OS will allocate to one process
  – other process will be denied
Avoiding Race Conditions

- How do we avoid a race condition?

- Need *mutual exclusion* (mutex):
  - ensure that if one process is using shared resource, another will be excluded

- Not all of process’s code will cause a race condition
  - e.g., when not accessing shared memory

- *Critical section* (critical region):
  - that part of the code where shared resource is accessed
Using Mutex

• To avoid race condition:
  – mutual exclusion to critical section
  – i.e., no two processes in critical section @ same time

• Enforcement of mutex creates additional problems

• *Deadlock*: each of two processes waiting for other
  – consider processes P1, P2 and resources R1, R2
  – OS allocates R1 to P1, R2 to P2

• *Starvation*: indefinitely denied access to resource
  – consider processes P1, P2, P3 and resource R
  – OS may alternate R between P1 and P3
  – P2 indefinitely denied

• Must avoid either of these!
Requirements for Mutual Exclusion

1. Mutual exclusion must be enforced
   (only one process in its CS @ a time)

2. Process that halts in non-CS does not interfere with other processes

3. No deadlock or starvation

4. When no process in its CS, any process requesting entry to CS must enter w/o delay

5. No assumptions about speed or # of CPUs

6. Process remains in CS for finite time only
Satisfying Mutex Requirements

• Approaches:
  – software solutions
    • processes must coordinate
    • no support from OS or PL
    • high overhead, prone to bugs
  – special-purpose machine instructions
    • low overhead
    • unattractive for general-purpose solution
  – provide support w/in OS
    • semaphores
    • monitors
    • message passing
Approaches to Mutex

- Process disables interrupts (hardware support)

```c
while (true) {
    /* disable interrupts */
    /* critical section */
    /* enable interrupts */
    /* remainder of non-CS */
}
```

- bad: unwise to give user processes this power
- doesn’t work in multiprocessor setting
- (however, kernel will often need to disable interrupts)
Approaches to Mutex

- Shared lock variable
  - initially 0
  - process checks value before entering CS
    - if value is 0, then sets to 1 and enters CS
    - if value is 1, waits until 0, then sets to 1, enters CS
  - again, a race condition — why?
Approaches to Mutex

- Pass control back and forth

Process P0:

```c
while (turn != 0) {
    /* sit&spin */;
    /* CRIT SECTION */
    turn = 1;
}
```

Process P1:

```c
while (turn != 1) {
    /* sit&spin */;
    /* CRIT SECTION */
    turn = 0;
}
```

- The “sit&spin” is called *busy waiting*: wastes CPU time
- Does this work?
Approaches to Mutex

- Pass control back and forth

Process P0:

```c
while (turn != 0) {
    /* sit&spin */
    /* CRIT SECTION */
    turn = 1;
}
```

Process P1:

```c
while (turn != 1) {
    /* sit&spin */
    /* CRIT SECTION */
    turn = 0;
}
```

- Guarantees mutual exclusion
- However, too lockstep: pace dictated by slower process

- **Spin lock**: lock that uses busy waiting
To Be Continued...

• More solutions next time:
  – correct software solution
  – solutions that don’t require busy wait
  – OS-provided mechanisms

• Now a Nachos-related topic...
Semaphores

• **Semaphore**: (Edsger Dijkstra 1965)
  – special variable that counts # of signals on it

• Two special ops:
  – `down()` [AKA `wait()` or `P()`]
    • P is for Dutch “proberen” -- test
  – `up()` [AKA `signal()` or `V()`]
    • V is for Dutch “verhogen” -- increment

• To xmit a signal via semaphore, process uses `V()`
• To receive signal via semaphore, uses `P()`
Semaphores

- View as variable w/ integer value with following three operations:
  - may be initialized to > 0 value
  - \( P() \) decrements the value; if value becomes < 0, invoking process blocks
  - \( V() \) increments the value; if value is not > 0, pick a process to unblock

- \( P() \) and \( V() \) assumed atomic

- A queue holds processes waiting on the semaphore
Semaphores

- For mutual exclusion, initialize semaphore to 1
- An example use of semaphore s to achieve mutual exclusion:
  \[
  \text{P}(s) ; \\
  /* \text{enter CRIT SECTION} */ \\
  \text{V}(s) ;
  \]

- At any time, value of semaphore can be interpreted as:
  - \( \geq 0 \): # of processes that can wait w/o blocking
  - \(< 0\): magnitude gives # of processes blocked in the queue

- However, can’t actually look at the value
  - \text{P}() \text{ and } \text{V}() \text{ are the only methods available}

- See Nachos: threads/sync.h \hspace{1cm} threads/sync.cc