CSE 120: Principles of Operating Systems
Lecture 5: Synchronization

Prof. Joseph Pasquale
University of California, San Diego

January 30, 2023

Synchronization

- Synchronize: events happen at the same time
- Process synchronization
  - Events in processes that occur "at the same time"
  - Actually, when one process waits for another
- Uses of synchronization
  - Prevent race conditions
  - Wait for resources to become available

The Credit/Debit Problem

- Say you have $1000 in your bank account
- You deposit $100
- You also withdraw $100
- How much should be in your account?
- What if deposit/withdraw occur at same time?

Credit/Debit Problem: Race Condition

Say P_0 runs first

Read \$1000 into b

Switch to P_1

Read \$1000 into b
Debit by \$100
Write \$900

Process P_0

```
Credit (int a) {
  int b;

  b = ReadBalance ();
  b = b + a;
  WriteBalance (b);

  PrintReceipt (b);
}
```

Process P_1

```
Debit (int a) {
  int b;

  b = ReadBalance ();
  b = b - a;
  WriteBalance (b);

  PrintReceipt (b);
}
```

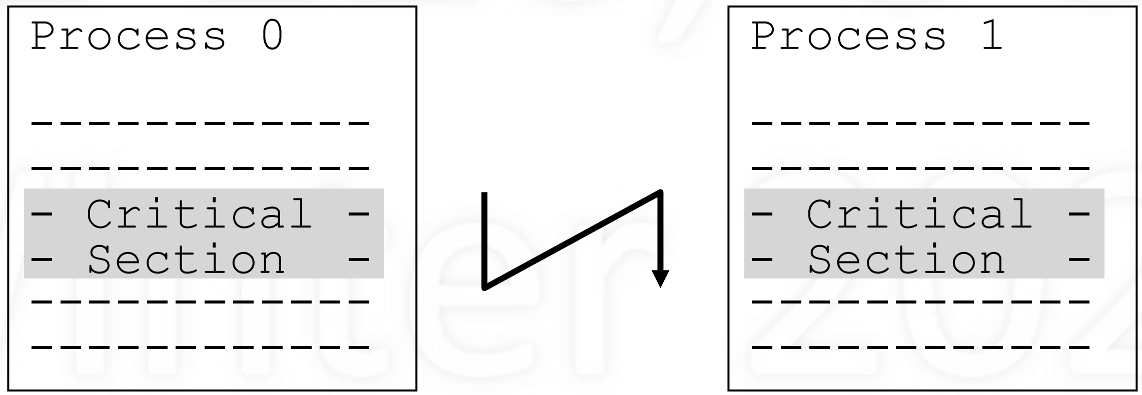
critical sections

Switch to P_0

Credit by \$100
Write \$1100

Bank just lost \$100!

To Avoid Race Conditions



- Identify related *critical sections*
  - Section(s) of code executed by different processes
  - Must run *atomically, with respect to each other*
- Enforce mutual exclusion
  - Only one process active in a critical section

What Does Atomic Really Mean?

- Atomic means “indivisible”
- We seek *effective* atomicity
  - can interrupt, as long as interruption has no effect
- It *is* OK to interrupt process in critical section
  - as long as other processes have no effect
- How to determine
  - Consider effect of critical section in isolation
  - Next consider interruptions: if same result, OK

How to Achieve Mutual Exclusion?

```
< entry code >                < entry code >
< critical section >          < critical section >
< exit code >                 < exit code >
```

- Surround critical section with entry/exit code
- Entry code should act as a barrier
  - If another process is in critical section, block
  - Otherwise, allow process to proceed
- Exit code should release other entry barriers

Requirements for Good Solution

- Given multiple cooperating processes
  - Each process has a critical section
  - All critical sections are to be mutually exclusive

1. At most one in a critical section at a time
2. Can't prevent entry if no others are in theirs
3. Should eventually be able to enter
4. No assumptions about CPU speed or number

Software Lock?

```
shared int lock = OPEN;
```

P_0

```
while (lock == CLOSED);
lock = CLOSED;
< critical section >
lock = OPEN;
```

P_1

```
while (lock == CLOSED);
lock = CLOSED;
< critical section >
lock = OPEN;
```

- Lock indicates if any process in critical section
- Is there a problem?

Take Turns?

```
shared int turn = 0;        // arbitrary set to P0
```

P_0

```
while (turn != 0);
< critical section >
turn = 1;
```

P_1

```
while (turn != 1);
< critical section >
turn = 0;
```

- Alternate which process enters critical section
- Is there a problem?

State Intention?

```
shared boolean intent[2] = {FALSE, FALSE};
```

P_0

```
intent[0] = TRUE;
while (intent[1]);
< critical section >
intent[0] = FALSE;
```

P_1

```
intent[1] = TRUE;
while (intent[0]);
< critical section >
intent[1] = FALSE;
```

- Process states intent to enter critical section
- Is there a problem?

Peterson's Solution

```
shared int turn;
shared boolean intent[2] = {FALSE, FALSE};
```

P_0

```
intent[0] = TRUE;
turn = 1;
while (intent[1] && turn==1);
< critical section >
intent[0] = FALSE;
```

P_1

```
intent[1] = TRUE;
turn = 0;
while (intent[0] && turn==0);
< critical section >
intent[1] = FALSE;
```

- If competition, take turns; otherwise, enter
- Is there a problem?
- There is a version for $n \geq 2$; more complex

What about Disabling Interrupts?

- Reasoning
  - No interrupts ⇒ no uncontrolled context switches
  - No uncontrolled context switches ⇒ no races
  - No races ⇒ mutual exclusion
- Is there a problem?

Test-and-Set Lock Instruction: TSL

- TSL mem (test-and-set lock: contents of mem)

```
do atomically (i.e., locking the memory bus)
     [ test if mem == 0 AND set mem = 1 ]
```

- Operations occur without interruption
  - Memory bus is locked
  - Not affected by hardware interrupts

What TSL Does, Expressed in C

- Assume C function, TSL(int *), that is *atomic*

```
TSL(int *lockptr)
{   int oldval;

    oldval = *lockptr
    *lockptr = 1;
    return ((oldval == 0) ? 1 : 0);
}
```

Effectively, this is what the TSL instruction does in one atomic instruction

Mutual Exclusion Using TSL

```
shared int lock = 0;
```

P_0

```
while (! TSL(&lock));
< critical section >
lock = 0;
```

P_1

```
while (! TSL(&lock));
< critical section >
lock = 0;
```

- Shared variable solution using TSL(int *)
  - tests if lock == 0 (if so, will return 1; else 0)
  - before returning, sets lock to 1
- Simple, works for any number of processes
- Still "suffers" from busy waiting

Semaphores

- Synchronization variable
  - Takes on integer values
  - Can cause a process to block/unblock
- wait and signal operations
  - wait (s) decrement; block if < 0
  - signal (s) increment; if any blocked, unblock one
- No other operations allowed
  - In particular, cannot test value of semaphore!

Examples and Interpretation

- wait (s) — decrement; block if < 0
- signal (s) — increment; if any blocked, unblock

- wait (1) — s $\rightarrow$ 0 — GO
- wait (0) — s $\rightarrow$ -1 — STOP (i.e., block)
- signal (-1) — s $\rightarrow$ 0 — GO *and* allow one to GO
- signal (0) — s $\rightarrow$ 1 — GO

Mutual Exclusion

```
sem mutex = 1;          // declare and initialize
```

P_0

```
wait (mutex);
< critical section >
signal (mutex);
```

P_1

```
wait (mutex);
< critical section >
signal (mutex);
```

- Use “mutex” semaphore, initialized to 1
- Only one process can enter critical section
- Simple, works for *n* processes
- Is there any busy-waiting?

Order How Processes Execute

```
sem cond = 0;
```

P_0

```
< to be done before P1 >
signal (cond);
```

P_1

```
wait (cond);
< to be done after P0 >
```

- Cause a process to wait for another
- Use semaphore indicating condition; initially 0
  - the condition in this case: "P_0 has completed"
- Used for ordering processes
  - In contrast to mutual exclusion

Semaphores: Only Synchronization

- Semaphores only provide synchronization
  - Synchronization: when a process blocks for event
- But, no information transfer
  - No way for a process to tell it blocked

Semaphore Implementation

- Semaphore s = [n, L]
  - n: takes on integer values
  - L: list of processes blocked on s
- Operations

```
wait (sem s) {
    s.n = s.n – 1;
    if (s.n < 0) add calling process to S.L and block; }
signal (sem s) {
    s.n = s.n + 1;
    if (s.L !empty) remove/unblock a process from s.L; }
```

Alternative Implementation

- Semaphore s = [n, L]
  - n: takes on integer values, non-negative
  - L: list of processes blocked on s
- Operations

```
wait (sem s) {
    if (s.n == 0) add calling process to s.L and block;
    else s.n = s.n – 1; }

signal (sem s) {
    if (s.L !empty) remove/unblock a process from s.L;
    else s.n = s.n + 1; }
```

Wait and Signal Must Be Atomic

- Bodies of wait and signal are critical sections
- So, still need mechanism for mutual exclusion!
- Use a lower-level (more basic) mechanism
  - Test-and-set lock
  - Peterson's solution
- So, busy-waiting still exists (can never remove)
  - But at lower-level (within semaphore operations)
  - Occurrence limited to brief/known periods of time

Analysis: Lower-Level Busy Waiting

- A calls wait (s), switch to B, B calls wait (s)
  - Switch occurs while A executing body of wait
- Body of wait is critical section, so B must block
  - Use test-set lock or Peterson's: busy waiting
- How long will B be blocked?
  - For time it takes to execute body of wait
- Small/known amount of time!
  - Compare to user critical section: unknown time

Are These Equivalent?

Implementation 1

```
wait (sem s) {
    s.n = s.n - 1;
    if s.n < 0 {
        addProc (me, s.L);
        block (me);
    }
}

signal (sem s) {
    s.n = s.n + 1;
    if (! empty (s.L)) {
        p = removeProc (s.L);
        unblock (p);
    }
}
```

Implementation 2

```
wait (sem s) {
    if s.n ≤ 0 {
        addProc (me, s.L);
        block (me);
    }
    s.n = s.n - 1;
}

signal (sem s) { // same
    s.n = s.n + 1;
    if (! empty (s.L)) {
        p = removeProc (s.L);
        unblock (p);
    }
}
```

Summary

- Synchronization: process waiting for another
- Critical section: code allowing race condition
- Mutual exclusion: one process excludes others
- Mutual exclusion mechanism: obey four rules
- Peterson's solution: all software, but complex
- Semaphores: simple flexible synchronization
  - wait and signal must be atomic, thus requiring lower-level mutual exclusion (Peterson's, TSL)

Textbook

- OSP: Chapter 5
- OSC: Chapter 5 (Process Synchronization)
  - Lecture-related: 6.1-6.6

Supplementary

- For those who wish to understand more subtle and advanced issues
- Will not be on exams

Mutual Exclusion Using TSL

- Critical section entry code

```
                             ; assume lock initially 0
loop:  TSL REG, lock         ; atomically {load REG with lock
                             ;         and store 1 into lock}
       CMP REG, #0           ; is REG (was lock) equal to 0?
       JNE loop              ; if not equal to 0, check again
                             ; also known as a “spin lock”
```

- Critical section exit code

```
       MOV lock, #0          ; reset lock to 0
```

Test and Test-and-Set

```
shared int lock = 0;

do {
    while (lock == 1);          // cheap
} while (! TSL(&lock));         // expensive
< critical section >
lock = 0;
```

- Busy-wait using simple reads of lock
  - Low overhead
- When lock opens, use test-and-set
  - Higher-overhead atomic operation less frequent

Efficient No-Spin Locking Code

- Entry code

```
while (! TSL(&lock)) {          // lock closed
      yield ();                 // give up CPU
}
```

- Exit code

```
lock = 0;                       // open lock
```

When is Busy-Waiting OK?

- Expected wait time < scheduling overhead
- Lots of processors (i.e., if waste is OK)
- Blocking is not an option (e.g., inside kernel)

How Costly is Busy Waiting?

- Consider time spent in critical section
  - Chance of context switch increases with length
  - If switch to process seeking entry, it will busy wait
  - Wastes an entire quantum – this is cost
- So, try to minimize time in critical section
- Compare critical sections that are
  - user code (e.g., application code)
  - system code (e.g., semaphore operations)