Synchronization Principles

CS 256/456

Dept. of Computer Science, University of Rochester

Recap of Last Class: CPU Scheduling

- CPU scheduling may take place at:
  - Hardware interrupt/software exception, system calls.
- Objectives:
  - Minimize completion time; maximize throughput
  - Minimize response time
  - Maintain fairness
- Policies:
  - FCFS, SJB, Priority
  - Round-Robin
  - Earliest Deadline First
- Multiple scheduling policies in system
- Linux 2.4 task scheduling

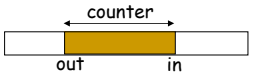
Synchronization Principles

- Background
  - Concurrent access to shared data may result in data inconsistency.
  - Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes.
- The Critical-Section Problem
  - Pure software solution
  - With help from the hardware
- Synchronization without busy waiting (with the support of process/thread scheduler)
  - Semaphore
  - Mutex lock
  - Condition variables

Bounded Buffer

- Shared data

```
typedef struct { ... } item;
item buffer[BUFFER_SIZE];
int in = 0, out = 0;
int counter = 0;
```



- Producer process

```
item nextProduced;
while (1) {
   while (counter==BUFFER_SIZE)
        ; /* do nothing */
   buffer[in] = nextProduced;
   in = (in+1) % BUFFER_SIZE;
   counter++;
}
```

- Consumer process

```
item nextConsumed;
while (1) {
   while (counter==0)
        ; /* do nothing */
   nextConsumed = buffer[out];
   out = (out+1) % BUFFER_SIZE;
   counter--;
}
```

Bounded Buffer

- The following statements must be performed *atomically*.

```
counter++;
counter--;
```

- Atomic operation means an operation that completes in its entirety without interruption.
- The statement "**counter++**" may be compiled into the following instruction sequence:

```
register1 = counter;
register1 = register1 + 1;
counter   = register1;
```

- The statement "**counter--**" may be compiled into:

```
register2 = counter;
register2 = register2 - 1;
counter   = register2;
```


Race Condition

- **Race condition**:
  - The situation where several processes access and manipulate shared data concurrently.
  - The final value of the shared data and/or effects on the participating processes depends upon the order of process execution.
- To prevent race conditions, concurrent processes must be **synchronized**.

The Critical-Section Problem

- Problem context:
  - *n* processes all competing to use some shared data
  - Each process has a code segment, called *critical section*, in which the shared data is accessed.
- Find a solution that satisfies the following:
  1. **Mutual Exclusion**. No two processes simultaneously in the critical section.
  2. **Progress**. No process running outside its critical section may block other processes.
  3. **Bounded Waiting/Fairness**. 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.

Eliminating Concurrency

- First idea: eliminating the chance of context switch when a process runs in the critical section.
  - software exceptions
  - hardware interrupts
  - system calls
- Disabling interrupts?
  - not feasible for user programs since they shouldn't be able to disable interrupts
  - feasible for OS kernel programs
    - for short critical sections
    - on single-processor machines

Critical Section for Two Processes

- Only 2 processes, P_0 and P_1
- General structure of process P_i (other process P_j)

```
do {
    entry section
        critical section
    exit section
        remainder section
} while (1);
```

- Processes may share some common variables to synchronize their actions.
- Assumption: instructions are atomic and no re-ordering of instructions.

Algorithm 1

- Shared variables:
  - `int turn;` initially `turn = 0;`
  - `turn==i` ⇒ P_i can enter its critical section
- Process P_i

```
do {
   while (turn != i) ;
      critical section
   turn = j;
      remainder section
} while (1);
```

- Satisfies mutual exclusion, but not progress

Algorithm 2

- Shared variables:
  - `boolean flag[2];` initially `flag[0] = flag[1] = false;`
  - `flag[i]==true` ⇒ P_i ready to enter its critical section
- Process P_i

```
do {
   flag[i] = true;
   while (flag[j]) ;
      critical section
   flag[i] = false;
      remainder section
} while (1);
```

- Satisfies mutual exclusion, but not progress requirement.

Algorithm 3

- Combined shared variables of algorithms 1 and 2.
- Process P_i

```
do {
   flag[i] = true;
   turn = j;
   while (flag[j] && turn==j) ;
      critical section
   flag[i] = false;
      remainder section
} while (1);
```

- Meets all three requirements; solves the critical-section problem for two processes. ⇒ called Peterson's algorithm.

Synchronization Using Special Instruction: TSL (test-and-set)

```
entry_section:
   TSL R1, LOCK        | copy lock to R1 and set lock to 1
   CMP R1, #0          | was lock zero?
   JNE entry_section   | if it wasn't zero, lock was set, so loop
   RET                 | return; critical section entered

exit_section:
   MOV LOCK, #0        | store 0 into lock
   RET                 | return; out of critical section
```

- Does it solve the synchronization problem?
- Does it work for multiple (>2) processes?
- What if you have special instruction **SWP** (swap the value of a register and a memory word)?

Solving Critical Section Problem with Busy Waiting

- In all our solutions, a process enters a loop until the entry is granted ⇒ busy waiting.
- Problems with busy waiting:
  - waste of CPU time
  - priority inversion
- What is the fundamental problem here?
- Develop solutions that do not busy wait. Need cooperation from the thread/process scheduler.

Semaphore

- Synchronization tool that does not require busy waiting.
- Semaphore *S* – integer variable which can only be accessed via two atomic operations
- Semantics (roughly) of the two operations:

```
wait(S) or P(S):
   wait until S>0;
   S--;

signal(S) or V(S):
   S++;
```

- Solving the critical section problem:

Shared data:

```
semaphore mutex=1;
```

Process Pi:

```
wait(mutex);
    critical section
signal(mutex);
    remainder section
```


Semaphore Implementation

- Define a semaphore as a record

```
typedef struct {
   int value;
   proc_list *L;
} semaphore;
```

- Assume two simple operations:
  - **block** suspends the process that invokes it.
  - **wakeup(P)** resumes the execution of a blocked process **P**.
- Semaphore operations now defined as (both are atomic):

```
wait(S):
   S.value--;
   if (S.value < 0) {
      add this process to S.L;
      block;
   }
signal(S):
   S.value++;
   if (S.value <= 0) {
      remove a process P from S.L;
      wakeup(P);
   }
```

Does this completely solve the critical section problem?
How to make sure wait(S) and signal(S) are atomic?
So have we truly removed busy waiting?

Mutex Lock

- Mutex lock – a semaphore with only two state: locked/unlocked
- Semantics of the two (atomic) operations:

  lock(mutex):

  wait until **mutex==unlocked;**
  mutex=locked;

  unlock(mutex):

  mutex=unlocked;

- Solving the critical section problem:

Shared data:

mutex=unlocked;

Process Pi:

lock(mutex);
critical section
unlock(mutex);
remainder section

Disclaimer

- Parts of the lecture slides contain original work of Abraham Silberschatz, Peter B. Galvin, Greg Gagne, Andrew S. Tanenbaum, and Gary Nutt. The slides are intended for the sole purpose of instruction of operating systems at the University of Rochester. All copyrighted materials belong to their original owner(s).