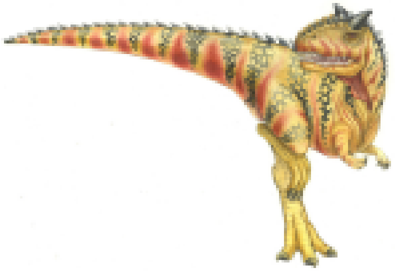
Chapter 6: SynchronizationTools



Outline

- Background

- The Critical-Section Problem
- Peterson's Solution
- Hardware Support for Synchronization
- Mutex Locks
- Semaphores
- Monitors
- Liveness
- Evaluation

Objectives

- Describe the critical-section problem and illustrate a race condition
- Illustrate hardware solutions to the critical-section problem using memory

barriers, compare-and-swap operations, and atomic variables

- Demonstrate how mutex locks, semaphores, monitors, and condition variables can be used to solve the critical section problem
- Evaluate tools that solve the critical-section problemin low-, Moderate-, and high-contention scenarios

Background

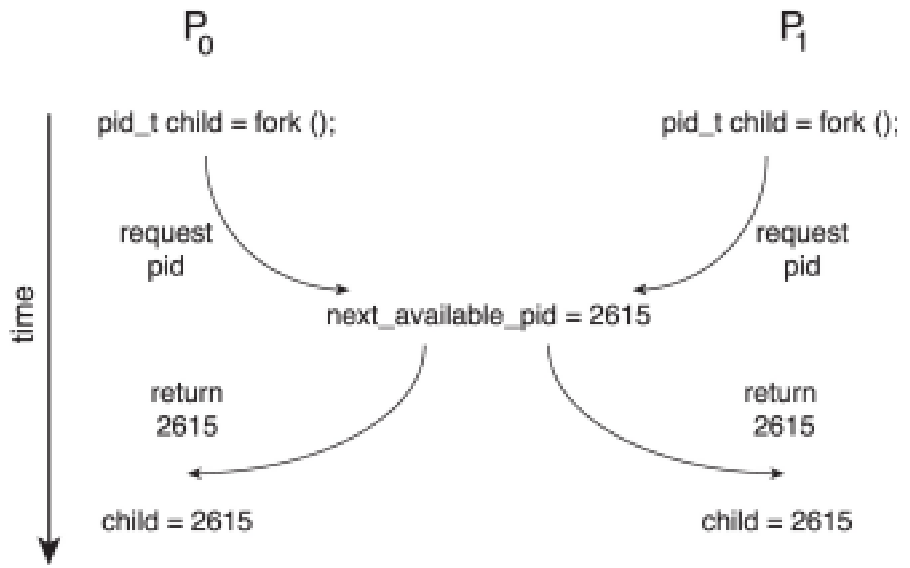
- Processes can execute concurrently
  - May be interrupted at any time, partially completingexecution
- Concurrent access to shared data may result in data inconsistency

- Maintaining data consistency requires mechanisms to ensuretheorderly execution of cooperating processes
- We illustrated in chapter 4 the problem when we consideredtheBounded Buffer problem with use of a counter that is updatedconcurrently by the producer and consumer,. which leadtoracecondition.

Race Condition

- Processes P_0 and P_1 are creating child processes usingthe `fork()` system call
- Race condition on kernel variable `next_available_pid`whichrepresents the next available

process identifier (pid)



- Unless there is a mechanism to prevent P_0 and P_1 fromaccessingthe variable `next_available_pid` the same pid could beassignedtotwo different processes!

Race Condition

- A situation where several processes access andmanipulatethesame data concurrently and the outcome of theexecutiondiffers from the particular order in which the accesstakesplace, is called a race condition.

- To guard against the race condition ensure only oneprocessata time can be manipulating the variable or data. Tomakesucha guarantee processes need to be synchronizedinsomeway.
- Critical section is one such solution

Critical SectionProblem

- 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, updatingtable, writing file, etc.
  - When one process in critical section, no other may beinitscriticalsection
- ***Critical section problem*** is to design protocol to solvethis▪ Each process must ask permission to enter critical sectionin**entrysection**, may follow critical section with **exit section**, theremainingcode is the **remainder section**

Critical Section

- General structure of process P_i

Critical-Section Problem (Cont.)

Requirements

for solution to critical-section problem1. **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 sectionandthereexist some processes that wish to enter their critical section, thentheselection of the process that will enter the critical sectionnext cannotbe postponed indefinitely

3. **Bounded Waiting** - A bound must exist on the number of timesthatother processes are allowed to enter their critical sectionsaftera process has made a request to enter its critical sectionandbeforethatrequest is granted

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

Interrupt-basedSolution

- Entry section: disable interrupts
- Exit section: enable interrupts
- Will this solve the problem?
  - What if the critical section is code that runs for an hour?
  - Can some processes starve – never enter their critical section.
  - What if there are two CPUs?

Software Solution1

- Two process solution
- Assume that the **`load`** and **`store`** machine-languageinstructions are atomic; that is, cannot be interrupted
- The two processes share one variable:
  - **`int turn; initlized to 0 (or 1)`**
- The variable **`turn`** indicates whose turn it is to enter thecritical section

Algorithmfor Process P_i

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

progress not satisfied: Eg: if turn ==0 and P1isreadytoenterits CS, P1 cannot do so, even though may be P0maybeinits remainder section

Software Solution2

- Replace variable turn with

  `boolean flag[2]`

- The **`flag`** array is used to indicate if a process is ready toenterthecritical section. Initialized to FALSE, indicates no one is interestedinentering the critical section
  - **`flag[i] = `*`true`*** implies that process P_i is ready!

```
do{
        flag[i] = true
            while (flag[j]);

            /* critical section */

            flag[i] = false;

            /* remainder section */
```

}while(1);

Peterson's Solution

- Two process solution
- Assume that the **load** and **store** machine-languageinstructions are atomic; that is, cannot be interrupted
- The two processes share two variables:
  - **int turn;**
  - **boolean flag[2]**

- The variable **turn** indicates whose turn it is to enter thecriticalsection
  - The **flag** array is used to indicate if a process isreadytoenter the critical section. Initialized to FALSE, initiallynooneisinterested in entering the critical section
    - **flag[i] = *true*** implies that process $\mathbf{P_i}$ isready!

Algorithmfor Process P_i

```
while (true){

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

          /* critical section */

        flag[i] = false;

        /* remainder section */

}
```

Correctness of Peterson’sSolution

- Provable that the three CS requirement are met:

  1. Mutual exclusion is preserved as only one process canaccessthe critical section at any time.

     $\mathbf{P_i}$ enters CS only if:

     either **`flag[j] = false`** or **`turn=i`**

  2. Progress requirement is satisfied as a process outsidethecritical section does not block other processes fromenteringthecritical section.

  3. Bounded-waiting requirement is met as every processgetsafairchance

Peterson's Solution and ModernArchitecture

- Disadvantages of Peterson's Solution
- It involves Busy waiting
- It is limited to 2 processes
- Although useful for demonstrating an algorithm, Peterson's Solution is not guaranteed to work on modern architectures.
  - To improve performance, processors and/or compilersmay reorder operations that have no dependencies

SynchronizationHardware

- Many systems provide hardware support for implementingthe critical section code.
  - Simple hardware instructions can be used effectively insolvingthecritical_x0002_section problem. These solutions are basedonthelocking —that is, protecting critical regions through theuseof locks.

```
while (true) {
```

```
acquire lock

   critical section

release lock

remainder section
}
```

Solution to Critical Section problem using locks

Hardware Instructions

- Modern machines provide special atomic hardware instructionsAtomic = non-interruptable
- Special hardware instructions that allow us to either *test-and-modify*the content of a word, or two *swap* the contents of two wordsatomically (uninterruptedly.)
  - **Test-and-Set** instruction

- **Swap** instruction

The test_and_set Instruction

- Definition

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

```
}
```

- Properties
  - Executed atomically
  - Returns the original value of passed parameter
    - Set the new value of passed parameter to **`true`**

Solution Usingtest_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);
```

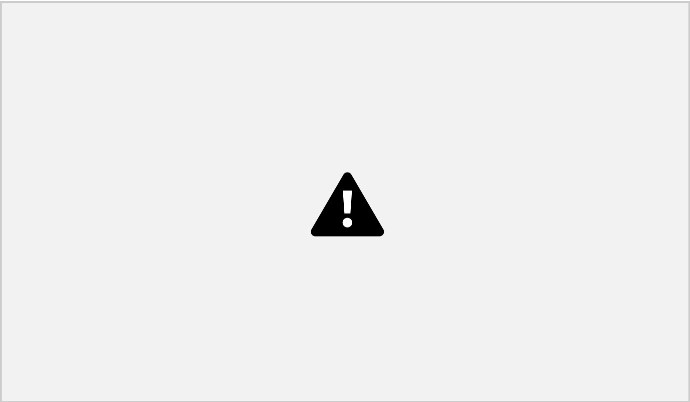

Solution Usingtest_and_set()

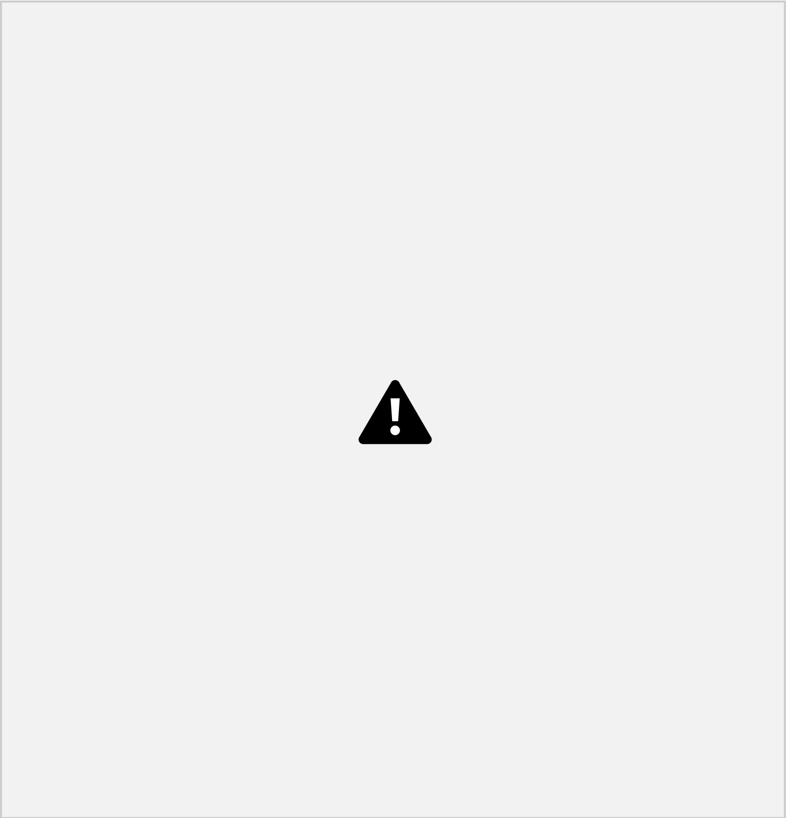
- X is a memory location associated with the CS and is initializedto 0.

The swap Instruction

- Using Swap() instruction, mutual exclusion can be providedas: AglobalBoolean variable lock is declared and is initialized to falseandeachprocess has a local Boolean variable key.

Definition of swap() function Mutual exclusion implementationwithSwap() instruction





Bounded-waitingwithtest-and-set

```
while (true) {
   waiting[i] = true;
   key = 1;
   while (waiting[i] && key == 1)
      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 = 0;
   else
      waiting[j] = false;
   /* remainder section */
}
```

Modern ArchitectureExample

- Two threads share the data:

```
boolean flag = false;
int x = 0;
```

- Thread 1 performs

```
while (!flag)
  ;
print x
```

- Thread 2 performs

```
x = 100;
flag = true
```

- What is the expected output?

100

Atomic Variables

- Typically, instructions such as compare-and-swap are usedas building blocks for other synchronization tools. ▪ One tool is an **atomic variable** that provides *atomic* (uninterruptible) updates on basic data types such as integers and booleans.
- For example:
  - Let **`sequence`** be an atomic variable
  - Let **`increment()`** be operation on the atomic variable**`sequence`**
  - The Command:

```
increment(&sequence);
```

ensures **`sequence`** is incremented without interruption:

Atomic Variables

▪ The **increment()** function can be implemented as follows:

```
void increment(atomic_int *v) {
     int temp;
     do {
                              temp = *v;
     }
     while (temp !=
 (compare_and_swap(v,temp,temp+1));}
```

Mutex Locks

- Previous solutions are complicated and generally inaccessibletoapplication programmers
- OS designers build software tools to solve critical sectionproblem
- Simplest is mutex lock
  - Boolean variable indicating if lock is available or not
- Protect a critical section by
  - First **`acquire()`** a lock

- Then **`release()`** the lock
  - Calls to **`acquire()`** and **`release()`** must be atomic
- Usually implemented via hardware atomic instructionssuchascompare-and-swap.
- But this solution requires **busy waiting** • This lock therefore called a **spinlock**

Solution to CS ProblemUsingMutexLocks

```
while (true) {
        acquire lock
```

```
            critical section

        release lock

    remainder section
}
```


Semaphore

- Synchronization tool that provides more sophisticated ways (than Mutex locks) for processes to synchronize their activities.
- Semaphore ***S*** – integer variable
- Can only be accessed via two indivisible (atomic) operations
  - **`wait()`** and **`signal()`**

4 Originally called `P()` and `V()`

- Definition of the `wait()` operation

```
wait(S) {
    while (S <= 0)
        ; // busy wait
    S--;
}
```

- Definition of the `signal()` operation

```
signal(S) {
    S++;
}
```


Semaphore(Cont.)

- **Counting semaphore** – integer value can range over an unrestricted domain
- **Binary semaphore** – integer value can range only between 0 and 1
  - Same as a **mutex lock**
- Can implement a counting semaphore ***S*** as a binary semaphore

- With semaphores we can solve various synchronization problems

Semaphore UsageExample

- Solution to the CS Problem
  - Create a semaphore "**mutex**" initialized to 1

```
wait(mutex);
   CS
signal(mutex);
```

- Consider P_1 and P_2 that with two statements S_1 and S_2 and the requirement that S_1 to happen before S_2
  - Create a semaphore "**synch**" initialized to 0

```
P1:
        S1;
        signal(synch);
    P2:
        wait(synch);
                                                  S2;
```


Semaphore Implementation

- Must guarantee that no two processes can execute the **`wait()`** and **`signal()`** on the same semaphore at the sametime
- Thus, the implementation becomes the critical sectionproblemwhere the **`wait`** and **`signal`** code are placed in the critical section
- Could now have **busy waiting** in critical section implementation
  - But implementation code is short
  - Little busy waiting if critical section rarely occupied

- Note that applications may spend lots of time in critical sectionsand therefore this is not a good solution

Semaphore ImplementationwithnoBusywaiting

- With each semaphore there is an associated waiting queue
- Each entry in a waiting queue has two data items:
  - Value (of type integer)
  - Pointer to next record in the list
- Two operations:
  - **block** – place the process invoking the operation ontheappropriate waiting queue
  - **wakeup** – remove one of processes in the waitingqueueand place

it in the ready queue

Implementation with noBusywaiting(Cont.)

- Waiting queue

```
typedef struct {
   int value;
              struct process *list;
} semaphore;
```

Implementation withnoBusywaiting(Cont.)

```
wait(semaphore *S) {
   S->value--;
   if (S->value < 0) {
                                add this process to S->list;
      block();
   }
```

```
}

signal(semaphore *S) {
   S->value++;
   if (S->value <= 0) {
      remove a process P from S->list; wakeup(P);
   }
}
```


Problems withSemaphores

- Incorrect use of semaphore operations:
  - `signal(mutex) …. wait(mutex)`
  - `wait(mutex) … wait(mutex)`

- Omitting of **`wait (mutex)`** and/or **`signal (mutex)`**

- These – and others – are examples of what can occur whensemaphores and other synchronization tools are used incorrectly.

Monitors

- A high-level abstraction that provides a convenient andeffective mechanism for process synchronization
- *Abstract data type*, internal variables only accessible by codewithin the procedure
- Only one process may be active within the monitor at atime
- Pseudocode syntax of a monitor:

```
monitor monitor-name
{
                        // shared variable declarations
  function P1 (…) { …. }

  function P2 (…) { …. }

  function Pn (…) {……}

                        initialization code (…) { … }

                                                   }
```


Schematic viewof aMonitor

ConditionVariables

- **`condition x, y;`**
- Two operations are allowed on a condition variable: • **`x.wait()`** – a process that invokes the operationissuspendeduntil **`x.signal()`**
  - **`x.signal()`** – resumes one of processes (if any) that invoked**`x.wait()`**
    - 4 If no **`x.wait()`** on the variable, then it has noeffect onthevariable

Monitor with ConditionVariables

Condition VariablesChoices

- If process P invokes **`x.signal()`**, and process Qis suspendedin**`x.wait()`**, what should happen next?
  - Both Q and P cannot execute in parallel. If Qis resumed, thenPmust wait
- Options include
  - **Signal and wait** – P waits until Q either leaves themonitor oritwaits for another condition
  - **Signal and continue** – Q waits until P either leaves themonitororit waits for another condition
  - Both have pros and cons – language implementer candecide•

Monitors implemented in Concurrent Pascal compromise

4 P executing signal immediately leaves the monitor, Qisresumed

- Implemented in other languages including Mesa, C#, Java

Monitor ImplementationUsingSemaphores

Variables

```
semaphore mutex; // (initially = 1) semaphore next; //
    (initially = 0) int next_count = 0; // number of
        processeswaitinginside the monitor
```

- Each function ***F*** will be replaced by

```
wait(mutex);
    …
  body of F;
    …
if (next_count > 0)
  signal(next)
else
  signal(mutex);
```

- Mutual exclusion within a monitor is ensured

Implementation– ConditionVariables

- For each condition variable ***x***, we have:

```
semaphore x_sem; // (initially=0)
int x_count = 0;
```

- The operation **`x.wait()`** can be implemented as:

```
x_count++;
if (next_count > 0)
    signal(next);
else
    signal(mutex);
wait(x_sem);
x_count--;
```

Implementation(Cont.)

- The operation `x.signal()` can be implemented as:

```
if (x_count > 0) {
    next_count++;
    signal(x_sem);
    wait(next);
    next_count--;
}
```

Resuming ProcesseswithinaMonitor

- If several processes queued on condition variable **`x`**, and **`x.signal()`** is executed, which process shouldbe resumed?
- FCFS frequently not adequate
- **conditional-wait** construct of the form **`x.wait(c)`** • Where **`c`** is **priority number**
  - Process with lowest number (highest priority) is scheduled next

Single Resourceallocation

- Allocate a single resource among competing processes usingprioritynumbers that specify the maximum time a process planstousethe resource

```
R.acquire(t);
    ...
  access the resurce;
    ...

  R.release;
```

- Where R is an instance of type **`ResourceAllocator`**

A Monitor to AllocateSingleResource

```
monitor ResourceAllocator
{
   boolean busy;
   condition x;
   void acquire(int time) {
         if (busy)
            x.wait(time);
         busy = true;
   }
   void release() {
         busy = FALSE;
         x.signal();
   }
   initialization code() {
    busy = false;
   }
}
```

Single ResourceMonitor(Cont.)

Usage:

```
acquire
  ...
release
```

- Incorrect use of monitor operations
  - **`release()`** … **`acquire()`**
  - **`acquire()`** … **`acquire()`**)
  - Omitting of **`acquire()`** and/or **`release()`**
- A process might never release a resource once it has beengrantedaccess to the resource. (Omitting of **`release()`**)
- A process might attempt to release a resource that it never requested.(Omitting of **`acquire()`**)

- A process might request the same resource twice (without first releasingthe resource). (**`acquire()`** … **`acquire()`**)

Endof Chapter6