Introduction & Process Synchronization

Dr. Kalpakis
Course URL: http://www.csee.umbc.edu/~kalpakis/courses/621

Topics of this course

- Process synchronization
- Process deadlocks
- Architectures of distributed systems
- Theoretical foundations of distributed systems
- Distributed mutual exclusion
- Distributed deadlock detection
- Distributed file systems
- Distributed shared memory
- Distributed scheduling
- Resource security and protection
- Queuing theory and system performance modeling
- Selected topics
Why Operating Systems are needed?

- Ever tried to use or manage a computer without an OS?
  - It is usually very
    - Cumbersome
    - Inefficient
    - Unfriendly
  - Thus, some S/W that eases the difficulties in using and managing computers will be very useful ($$$)

Functions of an Operating System

- Resource management
  - resources (CPU, memory, interrupts, disk, printers, displays, etc)
  - Need to help in managing a resource’s
    - Time & space
    - synchronization & deadlocks
    - accounting & state information

- User friendliness
  - Addresses issues related to
    - execution environment
    - error detection & handling
    - protection & security
    - fault-tolerance & recovery
Policies & Mechanisms

- Separate
  - policies, which describe what should be done
  - from mechanisms, which describe how policies are implemented
- In doing so, flexibility and adaptability
  - policies decisions be made at higher levels, and can be implemented using a small fixed set of mechanisms
  - policies can and do change over time as applications and users of resources change over time

Design Approaches

- Basic design approaches
  - Layered (Dijkstra)
    - vertically apportion OS functions to set of layers
    - Clearly defined interfaces and functionality for each layer
    - Example OS: Multics
  - Kernel (Hansen)
    - kernel=collection of primitives used by the rest of the OS
    - Contains at least process creation, control, and communication primitives.
    - Example OS: Hydra
  - Virtual Machine
    - provides the illusion to the user that all machine hardware is at this disposal
    - Example OS: VM/370 on IBM 370
Types of Advanced Operating Systems

- OS can be classified depending on whether they are
  - Architecture driven
    - Distributed operating systems
      - Concerned with lack of shared memory and physical global clock, and unpredictable communication delays
    - Multiprocessor operating systems
      - Shared memory accessed by multiple processors
  - Application driven
    - Database operating systems
    - Concurrency control, transaction management, and recovery
    - Real-time operating systems
      - Soft and hard real-time process scheduling (minimizing tardy processes)

Interacting Processes

- Process = program in execution

- Processes typically interact/cooperate with each other in order to provide various services

- Interaction can happen either by sharing a resource (e.g., memory location) or by exchanging messages

- Interacting processes need to coordinate in order to maintain the integrity constraints of the shared resources and to provide the intended service
**Processes**

- A process during its lifetime goes can be in any of these states
  - running
  - ready
  - blocked
- concurrent processes=their lifetimes overlap
- serial processes= their lifetimes (execution time period) do not overlap
- processes interact via shared variables or messages
- Process state information includes program counter, page tables, state, priorities, privileges, open file descriptors, pending I/Os, activation stack, register contents, etc
- Process switching/swapping is expensive
- threads are lightweight processes(execution threads) within a process which share common state information between them
- Threads have lower cost of switching/management than processes

---

**Critical Section Problem**

- Critical section
  - a code segment in which a shared resource is accessed
- In order to maintain integrity of shared resource mutual exclusion is needed
- The mutual exclusion problem (Mutex)
  - requirements
    - only 1 process is executing in CS
    - when no process in CS any process can get in CS in finite time
    - No process can prevent any other process to enter CS indefinitely
- Early mechanisms for Mutex
  - busy waiting
  - disabling interrupts
A Solution to the Critical Section Problem (Peterson’81)

```c
#define FALSE   0
#define TRUE   1
#define N      2                       /* number of processes */
int turn;                               /* whose turn is it? */
int interested[N];                      /* all values initially 0 (FALSE) */

void enter_region(int process)  {        /* process is 0 or 1 */
  int other;                                 /* number of the other process */
  other = 1 - process;                  /* the opposite of process */
  interested[process] = TRUE;       /* show that you are interested */
  turn = process;                /* set flag */
  while (turn == process &&
    interested[other] == TRUE) ;                                               /* wait */
}

void leave_region(int process)  {        /* process: who is leaving */
  interested[process] = FALSE;     /* indicate departure from critical region */
}
```

Semaphores (Dijkstra’65)

- A semaphore is
  - an integer variable S with an associated queue and two operations
    - P(S)
      - if S >= 1 the S := S-1 else wait (block) on semaphore’s queue
    - V(S)
      - if queue not empty unblock a waiting process else S := S +1
  - Binary and integer semaphores (depending on the initial value of S)
  - Mutex is achieved with binary semaphores S
    - P(S);
    - Critical Section Code;
    - V(S);
  - Drawbacks of semaphores
    - processes need to know which other processes use the semaphores and how
    - P & V operations must be placed carefully
    - difficult to debug and verify
Some Famous Synchronization Problems

- Dining Philosophers
- Producer-Consumer
- Readers-Writers
  - reader’s priority: arriving readers have priority over waiting writers
  - writer’s priority: an arriving writer has priority on waiting readers; an arriving/waiting reader gets in CS only if there not writers in the system
  - starvation
  - weak readers/writers
- Sleeping Barber
- Cigarette smokers

Monitors (Hoare’74)

- ADT for defining shared objects and methods to access them
- Properties
  - only one process can be active within a monitor at a time; all other processes are placed in an entry queue
  - processes that are active within a monitor can only access local data
  - monitor data can not be accessed from outside monitor
- Special operations
  - wait
    - active process joins a monitor queue and relinquishes control of the monitor
  - signal
    - signaling process joins the urgent queue and allows a waiting process to gain control/entry into the monitor
    - urgent queue has higher priority than the entry queue
Monitors

- **Condition variables are monitor special variables**
  - enable wait and signal operations to be distinguished and hence provide finer synchronization
  - have an associated queue where processes can wait
  - Provide operations for processes to execute
    
    `<cond-var>.wait([<priority>]);`
    `
    <cond-var>.signal;`
    `
    <cond-var>.queue;`

Monitor Program Skeleton

```
Monitor-name: monitor;

  begin
    <ordinary variable declarations>
    <cond-var>: condition;
    procedure <proc-name>;
      begin;
      … <cond-var>.<signal or wait> …
      … if <cond-var>.queue then ….
      End;
      begin
        <init-code>
      end;
```

**Drawbacks of Monitors**

- no concurrency if shared resource is encapsulated in the monitor
  - if not, then monitor methods need to be called before & after accessing shared resource
  - integrity of shared resource is not ensured

- nested monitor calls can lead into deadlocks
  - since a wait in an inner monitor relinquishes control of that monitor only, but not the outer monitor

**Serializers (Hewitt & Anderson’79)**

- Serializers are similar to monitors
  - all events that gain/release control are totally ordered in time
  - only one processes is active in a serializer
  - provide hollow regions where multiple processes can be active
  - offer automatic signaling where a process specifies the resume condition at the time it makes the wait call

- processes entering a hollow region relinquishes control of the serializer

\[
\text{join-crowd}(\text{<crowd>}) \text{ then } \text{<body>} \text{ end}
\]

  where <crowd> maintans the list of processes (ids) active in hollow region

- empty(<crowd>) tests whether any process is active in <crowd>
Serializer Queues

- Serializers have queue variables instead of condition variables
  
  `enqueue(<priority>, <queue-name>) until <condition>`

  - the <condition> is checked when a process reaches the head of the priority queue; if condition is true and no one is in the serializer, the waiting process regains control of serializer
  - `empty(<queue>)` tests whether <queue> is empty

- All events that gain/release control are totally ordered in time
  
  - entry and exit events
  - guarantee, establish, and timeout events (queue related)
  - join-crowd and leave-crowd events

Solving the Readers/Writers problem using Serializers

```
serializer ReadersWriters {
    queue readq, writeq;
    crowd rcrowd, wcrowd;
    database db;

data read(int key) {
    data x;
    enqueue(readq) until empty(wcrowd);
    joincrowd(rcrowd) {
        x = readDB(db, key);
    };
    return x;
};

void write(data x, int key) {
    enqueue(writeq) until
        empty(wcrowd) && empty(rcrowd) && empty(readq);
    joincrowd(wcrowd) { writeDB(db, x, key); };
};
}
```
Drawbacks of Serializers

- Less efficient due to complexity and automatic signalling

Path Expressions (Campbell & Habermann’74)

- A path expression
  - Is similar to a regular expression (with special operators) with alphabet the set of operations on the shared resource
  - Restricts the set of admissible execution histories of operations on the shared resource
  - Indicates interleave order of shared resource operations
Path Expressions

Path expression path <S> end, where <S> is an expression with operands resource operations and the following three operators

- **Sequencing (;)**
  - path open; read; close; end

- **Selection (+)**
  - path read + write end

- **concurrency ({}**
  - path write; {read} end
  - path {write; read} end

- If there are more than one path expressions then an admissible execution histories must satisfy all of them
  - path startRead + {startWrite; write} end
  - path {startRead; read} + write end

Communicating Sequential Processes (Hoare’78)

- I/O commands can been treated as synchronization primitives
- In CSP processes communicate via I/O commands
  - I/O commands are synchronous
  - each I/O command specifies the name of the target input or output process
  - an input and an output command correspond if type of output agrees with type of input variable
  - I/O commands execute if they have matching source/destination processes and they correspond
CSP Syntax

- **CSP commands**
  - **input command**
    
    `<source process name> ? <target variable>`
  
  - **output command**
    
    `<destination process name> ! <expression>`
  
- **Concurrency**
  
    `[<process code> || … || <process code>]`
  
- **guarded command**
  
    `<guard expression E> --> <command list L>`
    
    - where E is a boolean expression (including an input command)
    - a guarded command succeeds (i.e. L is executed) only if its guard evaluates to true
  
- **alternative command**
  
    `[ <guarded command> # … # <guarded command>]`
  
- **repetitive command**
  
    `*[^<alternative command>]`

Solving the Producer/Consumer Problem with CSP

```csp
process buffer {
    buf pool[0..9];
    int in=0, out=0;
    *[ in < out+10 && producer?pool(in mod 10) --> in++; #
        out < in && consumer?more() --> consumer!pool[out mod 10]; out++
    ]
}
process producer {
    buf p;
    while (true) {
        produce();
        buffer!p;
    }
}
process consumer{
    buf q;
    while(true){
        buffer!more();
        buffer?q;
        consume(q);
    }
}
```
Drawbacks of CSP

- explicit naming of processes in I/O commands
- introduces delays and inefficiencies due to the blocking I/O commands