

Reference Papers: Recommend reading appropriate reference papers given in each chapter end.
Contact Information

B. Prabhakaran
Department of Computer Science
University of Texas at Dallas
Mail Station EC 31, PO Box 830688
Richardson, TX 75083
Email: praba@utdallas.edu
Fax: 972 883 2349
URL: http://www.utdallas.edu/~praba/cs6378.html
Phone: 972 883 4680
Office: ES 3.706
Office Hours: 3 - 4 pm. Mondays, Wednesdays.
Other times by email appointments
Announcements: Made in class and on course web page.
TA: TBA.
Course Outline

Proposed Outline. Might be modified based on time availability:

- Introduction to operating systems, inter-process communication. (Chapters 1 & 2).
- Distributed Operating Systems
  - Architecture (Chapter 4)
  - Clock Synchronization, Ordering (Chapter 5)
  - Distributed Mutual Exclusion (Chapter 6)
  - Distributed Deadlock Detection (Chapter 7)
  - Agreement Protocols (Chapter 8)
- Distributed Resource Management
  - Distributed File Systems (Chapter 9)
  - Distributed Shared Memory (Chapter 10)
  - Distributed Scheduling (Chapter 11)
Course Outline ...

- Recovery & Fault Tolerance
  - Chapters 12 and 13
- Concurrency Control/ Security
  - Depending on time availability

Discussions will generally follow the main text. However, additional/modified topics might be introduced from other texts and/or papers. References to those materials will be given at appropriate time.
Evaluation

- 2 Mid-terms: in class. 75 minutes. Mix of MCQs (Multiple Choice Questions) & Short Questions.
- 1 Final Exam: 2 hours. Mix of MCQs and Short Questions.
- 2 - 3 Quizzes: 5-6 MCQs. 15 minutes each.
- Homworks/assignments: 3 or 4 spread over the semester.
- Programming Projects: 2 - 3 spread over the semester
Grading

- Home works: 5%
- Quizzes: 5%
- Mid-terms: 40% (20% each)
- Final Exam: 25%
- Projects: 25%
Schedule

- Quizzes: Dates announced in class & web, a week ahead.
- Mid-term 1: October 2, 2002
- Mid-term 2: November 11, 2002
- Final Exam: To be announced
- Subject to minor changes
- Projects and homework schedules will be announced in class and course web page, giving sufficient time for submission.
Programming Projects

- No copying/sharing of code/results will be tolerated. Any instance of cheating in projects/homeworks/exams will be reported to the University.
- Individual projects.
- Each project will be for 25 marks.
- Projects might involve Unix, C/C++ programming, network programming.
- Deadlines will be strictly followed for projects and homeworks submissions.
- Projects submissions through Web CT.
- Demo may be needed
Web CT

- Go to: [http://webct.utdallas.edu](http://webct.utdallas.edu)
- Web CT has a discussion group that can be used for project and other course discussions.
Cheating

- Academic dishonesty will be taken seriously.
- Cheating students’ will be handed over to Head/Dean for further action.
- Remember: home works/projects (exams too !) are to be done individually.
- Any kind of cheating in home works/ projects/ exams will be dealt with as per UTD guidelines.
- Cheating in any stage of projects will result in 0 for the entire set of projects.
Projects

- 2 - 3 projects planned.
- Involves exercises such as ordering, deadlock detection, load balancing, message passing, and implementing distributed algorithms (e.g., for scheduling, etc.).
- Platform: Linux, C/C++. Network programming will be needed. Multiple systems might be used.
- Specific details and deadlines will be announced in class and course webpage.
- Suggestion: Learn network socket programming and threads, if you do not know already. Try simple programs for file transfer, talk, et.
- Sample programs and tutorials available at:
  - http://www.utdallas.edu/~praba/projects.html
Homeworks

- Each homework will be for 10 marks.
- Homeworks Submission:
  - Submit on paper to TA/Instructor.
Basic Computer Organization

- Input Unit
- Output Unit
- CPU
- Memory
- ALU (Arithmetic & Logic Unit)
- Secondary Storage
Simplified View of OS

Physical Memory

Virtual Memory

Memory Space

OS CEKernel

OS Tools

User Processes

User Processes ..

Tools ++

User i
Process j

Code
Data

Code
Data

Code
Data

Code
Data
Distributed View of the System
Inter-Process Communication

- Need for exchanging data/messages among processes belonging to the same or different group.
- IPC Mechanisms:
  - Shared Memory: Designate and use some data/memory as shared. Use the shared memory to exchange data.
    - Requires facilities to control access to shared data.
  - Message Passing: Use “higher” level primitives to “send” and “receive” data.
    - Requires system support for sending and receiving messages.
  - Operation oriented language constructs
    - Request-response action
    - Similar to message passing with mandatory response
    - Can be implemented using shared memory too.
IPC Examples

- Parallel/distributed computation such as sorting: shared memory is more apt.
  - Using message passing/RPC might need an array/data manager of some sort.

- Client-server type: message passing or RPC may suit better.
  - Shared memory may be useful, but the program is more clear with the other types of IPCs.

- RPC vs. Message Passing: if response is not a must, atleast immediately, simple message passing should suffice.
Shared Memory

Writers/Producers

Only one process can write at any point in time. No access to readers.

Shared Memory

Readers/Consumers

Multiple readers can access. No access to Writers.
Shared Memory: Possibilities

- Locks (unlocks)
- Semaphores
- Monitors
- Serializers
- Path expressions
Locks as Language Constructs

- Algol like constructs
- Cobegin-coend
  - resource ri(variable list), …, rn(variable list);
  - cobegin S1 || S2 || … || Sn coend
- With-when
  - with r when B do S
  - r: resource; B: boolean condition; S: action statement
- 2 or more executions of with statement on the same resource is serialized.
Semaphores

**P(share):**
- share < 1

**V(share):**
- if queue > 1, else share = share + 1

**Invariant:** No. of Ps – No. of Vs <= initial value

**Initial Value:** share = 1
Semaphore: Example

Shared var
mutex: semaphore (=1);

Process I (I=1,n);
begin

.....
P(mutex);
execute CS; // Critical Section
V(mutex);
.....
End.
Reader’s Priority Problem

- If no reader/writer present:
  - Read Data
  - Control back to readers, if any

- If readers present:
  - New Reader
    - If no readers:
      - If NO reader:
        - Control back to writers, if any
    - Give control back to readers

- New Writer
  - If NO reader:
    - Control back to writers, if any
Reader’s Priority: Semaphores

Shared var
nreaders: integer;
mutex, wmutex, srmutex: semaphore;

procedure reader;
begin
  P(mutex);
  if nreaders = 0 then nreaders := nreaders + 1;
P(wmutex)
  else nreaders := nreaders + 1;
 V(mutex);
  read(f);
P(mutex);
  nreaders := nreaders - 1;
  if nreaders = 0 then V(wmutex);
  V(mutex);
end.

procedure writer(d: data);
begin
  P(srmutex);
P(wmutex);
  write(f, d);
V(wmutex);
V(srmutex);
end.

begin (* initialization *)
mutex = wmutex = srmutex = 1;
 nreaders := 0;
end.
Semaphore: Drawbacks

- Development of semaphore based programs must be coordinated closely. All cooperating processes should know how semaphores are being used.
- Mistakes such as missing a P or V can result in inconsistent access, deadlocks.
- Difficult to verify correctness. Why?
Reader’s Weak Priority

If NO reader/writer

Random choice of reader/writer

Read Data

File System

Write Data

Reader

Give control back to readers

If NO reader

New Reader

If readers present

New Writer

Writer

Reader’s Weak Priority

File System

New Reader

If NO reader/writer

Read Data

Random choice of reader/writer

Write Data

Reader

Give control back to readers

If NO reader

New Writer

If readers present

Writer

B. Prabhakaran

26
Monitors

- Only one process can be active within the monitor at a time.
  - An implicit process associated with the monitor ensures this. When one process is active in a monitor, other processes are put in a queue.
- Procedures in a monitor can access data local to the monitor: cannot access an outside variable.
- Variables/data local to a monitor cannot be directly accessed from outside.
Monitors ..

Process a

ConditionVariable.wait

Procedure i

ConditionVariable.signal

Queue for each conditional variable

Only one process at a given time

Procedure j

Process b

Data
Monitor Structure

<Monitor-name> monitor begin
Declaration of data local to the monitor
.....
procedure <Name> (<formal parameters>); begin
procedure body
end.

Declaration of other procedures ..
begin
local data initialization
end;
end.

B. Prabhakaran
Designing a Monitor

- When a process is active inside a monitor, other processes get queued up.
- Queuing up: done by operation *wait*. Dequeuing by *signal*.
- Queuing/dequeuing up: more than 1 reason possible. E.g., waiting for a reader to depart, waiting for a writer to finish, ..
- Condition variable may be associated with wait and signal. E.g., OKtoread.wait, OKtoread.signal, ...
- Queues: generally FIFO, priorities may be implemented with a parameter.
Reader’s Priority: Monitors

readers-writers: monitor;
begin
    readercount: integer;
    busy: boolean;
    OKtoread, OKtowrite: condition;
procedure startread;
    begin
        if busy then OKtoread.wait;
        readercount := readercount + 1;
        OKtoread.signal; // all readers can start
    end startread;
procedure endread;
    begin
        readercount := readercount - 1;
        if readercount = 0 then OKtowrite.signal;
    end endread;
procedure startwrite
begin
  if busy OR readcount != 0 then
    OKtowrite.wait;
  busy := true;
end startwrite;

procedure endwrite
begin
  busy := false;
  if OKtoread.queue then
    OKtoread.signal;
  else OKtowrite.signal;
end endwrite;
begin // initialization
  readercount := 0; busy := false;
end;
end readers-writers.
Readers-Writers: Monitors

Reader:
   startread();
   readfile();
   endread();

Writer:
   startwrite();
   writefile();
   endwrite();
Monitors: Drawbacks

- Only one active process inside a monitor: no concurrency.
- Previous example: File NOT inside monitor to allow concurrency. -> Responsibility of readers and writers to ensure proper synchronization.
- Nested monitor calls can lead to deadlocks:
  - Consider monitors X and Y with procedures A and B. Let X.A call Y.B and vice-versa
  - A process P calls X.A, process Q calls Y.B.
  - P is blocked on Y.B and Q is blocked on X.A
- Responsibility of valid programs shifts to programmers, difficult to validate correctness.
Serializers

Only 1 active process

Multiple active processes

Enqueue (<priority>,<qname>)
until (<condition>)

join-crowd (<crowd>) then
(<body>) end
Reader’s Priority: Serializers

Readerwriter: serializer

var
  readq: queue; writeq: queue; rcrowd: crowd; wcrowd: crowd; db: database;
procedure read(k:key; var data: datatype);
begin
  enqueue (readq) until empty(wcrowd);
  joincrowd (rcrowd) then
    data:= read-db(db[key]);
end return (data);
end read;

procedure write(k:key, data:datatype);
begin
  enqueue(writeq) until
    (empty(rcrowd) AND empty(wcrowd) AND empty(readq));
  joincrowd (wcrowd) then write-db(db[key], data);
end
end write;
Readers-Writers in Serializers ...

- Weak reader’s priority
  - enqueue(writeq) until (empty(wcrowd) AND empty(rcrowd));
  - A writer does not wait until readq becomes empty

- Writer’s priority
  - enqueue(writeq) until (empty(wcrowd) AND empty(rcrowd));
  - enqueue(readq) until (empty(wcrowd) AND empty(writeq));
Serializers: Drawbacks

- More complex, may be less efficient
- More work to be done by serializers
- “crowd”: complex data structure; stores identity of processes,…
- Assumes automatic signaling feature: test conditions of every process at the head of every queue every time a process comes out of a serializer.
- Though it (automatic signalling) helps in avoiding deadlocks and race conditions.
Message Passing

- Blocked Send/Receive: Both sending and receiving process get blocked till the message is completely received. Synchronous.
- Unblocked Send/Receive: Both sender and receiver are not blocked. Asynchronous.
- Unblocked Send/Blocked Receive: Sender is not blocked. Receiver waits till message is received.
- Blocked Send/Unblocked Receive: Useful?
- Can be implemented using shared memory. Message passing: a language paradigm for human ease.
**Un/blocked**

- **Blocked message exchange**
  - Easy to understand, implement, verify correctness
  - Less powerful, may be inefficient as sender/receiver might waste time waiting

- **Unblocked message exchange**
  - More efficient, no waste on waiting
  - Needs queues, i.e., memory to store messages
  - Difficult to verify correctness of programs
Message Passing: Possibilities
Message Passing: Possibilities...
Naming

- **Direct Naming**
  - Specify explicitly the receiver process-id.
  - Simple but less powerful as it needs the sender/receiver to know the actual process-id to/from which a message is to be sent/received.
  - Not suitable for generic client-server models

- **Port Naming**
  - Receiver uses a single port for getting all messages, good for client-server.
  - More complex in terms of language structure, verification

- **Global Naming (mailbox)**
  - Suitable for client-server, difficult to implement on a distributed network.
  - Complex for language structure and verification
Commn. Seq. Processes (CSP)

- A language that incorporates constructs for sending & receiving messages.
- Send & receive: synchronous -> Processes coordinate and synchronize when communication occurs.
- Naming & Typing:
  - receive: input command specifies explicitly the process (sender) from which it wants the message.
  - Send: output command specifies explicitly the process (receiver) to which the message is to be delivered.
  - Target variable’s type in receiver should match message type.
- Concurrency among processes: [ process Pi || Pj ||… ].
Guards in CSP

- **G -> CL.** G is the guard condition. When true, commands in the list CL are executed.

- **G1 -> CL1 G2 -> CL2 G3 -> CL3 …**
  - If G1 is true, then CL1; else if G2 is true, then CL2; ..
  - If more than one guard is true, one of them is chosen randomly.
  - If none of the guards are true, the entire set of “alternative”/selection commands fail.
  - Guards can be combined with input commands -> A process need not wait on the message to be available. Helps in asynchronous message exchange.

- **Repetition: *[ ….. ]**
CSP: Example

process reader-writer
    OKtoread, OKtowrite: integer (initially = value);
    busy: boolean (initially = 0);
    *
        busy = 0; writer?request() ->
            busy := 1; writer!OKtowrite;
        *
        busy = 0; reader?request() ->
            busy := 1; reader!OKtoread;
        *
        busy = 1; reader?readfin() -> busy := 0;
        *
        busy = 1; writer?writefn() -> busy := 0;
    ]
CSP: Drawbacks

- Requires explicit naming of processes in I/O commands.
- No message buffering; input/output command gets blocked (or the guards become false) -> Can introduce delay and inefficiency.
Operation oriented constructs

Remote Procedure Call (RPC):

Task A

Service declarations

......
RPC: abc(xyz, ijk);
......

send xyz; wait for result

Task B

Service declarations

......

return ijk

- Service declaration: describes in and out parameters
- Can be implemented using message passing
- Caller: gets blocked when RPC is invoked.
- Callee implementation possibilities:
  - Can loop “accepting” calls
  - Can get “interrupted” on getting a call
  - Can fork a process/thread for calls
Pointer passing, global variables passing can be difficult.

If processes on different machines, data size (number of bits for a data type) variations need to be addressed.

- Abstract Data Types (ADTs) are generally used to take care of these variations.
- ADTs are language like structures that specify how many bits are being used for integer, etc…
- What does this imply?

Multiple processes can provide the same service? Naming needs to be solved.

Synchronous/blocked message passing is equivalent to RPC.
**Ada**

```
task [type] <name> is
  entry specifications
end
```

Task body <name> is
Declaration of local variables
begin
  list of statements
  ...
  **accept** <entry id> (<formal parameters> do
  body of the accept statement
  end<entry id>
  **exceptions**
  Exception handlers
end;

```
task proc-buffer is
  entry store(x:buffer);
  remove(y:buffer);
end;
```

```
task body proc-buffer is
  temp: buffer;
  begin
    loop
      accept store(x: buffer);
      temp := x;
      end store;
      accept remove(y:buffer);
      y := temp;
      end remove;
    end loop
  end proc-buffer.
```
Ada Message Passing

Somewhat similar to executing procedure call. Parameter value for the entry procedure is supplied by the calling task. Value of Result, if any, is returned to the caller.
Ada Guards

- **when** <condition>: similar to CSP guard. Can be combined with accept statements.
- Multiple guarded statements can be randomly chosen: **select** construct.
- All guard statements are evaluated to determine “open” accept statements.
- Random selection in case of multiple open accept statements.
- If no accept statement is open and no other alternative statement can be executed, exception is raised.
An Example

loop
    select
        when G1 -> accept P1 do .... End;
        or ....
        or ....
        delay (5.0);
        else null;
    end select
end loop;

- If no accept statement is ready, then delay statement will be executed.
- After the delay statement, check the accept statements again.
- If no accept statement is ready, exit the select statement and continue on next loop.
- Delay of 0.0 -> becomes a simple polling statement.
- When no immediate message exchange is possible, else part is executed.