

## Exam 3 Review

### Exam location and time:

- Tuesday, May 13th, GS-906
- 8:00 - 10:00 am
- May use two hand-written pages (no larger than 8.5 x 11 inches) of notes.

### Monitors:

- Shared memory mechanism.
- Implicit synchronization, explicit signaling.
- Condition variables: **wait**, **signal**, **signal\_all**, **empty**.
- Signal semantics:
  - Signal-and-continue: signaler continues executing, signaled process gets placed on entry queue.
  - Signal-and-wait: signaler blocks, signaled process gets to execute immediately, signaler process gets placed on entry queue.
- Techniques:
  - Pass the condition — “hand off” the condition from signaler to signaled process.
  - Covering condition — one condition variable to “cover the conditions that processes test. When signaled, the process re-checks its specific condition.

### Monitors, (continued):

- Problems:
  - Shortest-job-next.
  - One-lane bridge.
  - Disk scheduling algorithms — separate scheduler, scheduler as intermediary, nested monitor.
  - Bear-and-honey-bees problem.
- Languages — general idea on exam, not specifics:
  - pthreads:
    - **pthread\_mutex\_init**, **pthread\_mutex\_lock**, **pthread\_mutex\_unlock**.
    - **pthread\_cond\_init**, **pthread\_cond\_wait**, **pthread\_cond\_signal**, **pthread\_cond\_broadcast**.
  - Java:
    - **synchronized** — applied to code segments or methods.
    - Locks object, not just the specific method.
    - One lock per object.
    - **wait**, **notify**, **notify\_all** operations.
    - One condition variable per object; thus, no condition variable needed as argument.
- Why does the pthreads implementation need the mutex\_lock operations? The Java version does not have these.

**Message Passing:** Chapter 7, notes: 7-Chapter7.pdf

- Synchronization plus data.
- Channel: a generic message structure.
  - Generally a queue of messages.
  - Tag or type field can be used by receive to “choose” messages from the queue.
- **send** and **receive**:
  - Synchronous: both block.
  - Asynchronous: **receive** blocks, **send** does not.
  - Most implementations provide blocking and non-blocking **send**, and a non-blocking conditional **receive**.
- Interaction patterns:
  - Filters: one-way data flow.
  - Client-Server: server listens, client initiates contact, two-way data flow, asymmetric.
    - Server may “hold onto” a request and not reply to it until “later”.
    - Multiple clients/one server, multiple clients/many servers.
    - Servers can call other servers.
  - Interacting peers: two-way data flow, symmetric.
    - Central server, symmetric (fully-connected graph), ring.

**Message Passing, (continued):**

- Implementations — basic ideas:
  - MPI
    - Multiple Unix processes, possibly on different hosts.
    - send and receive primitives; blocking receive (there is also a non-blocking receive).
    - message tags.
    - broadcast, barrier, and reduction operations available.
  - Java and C
    - **ServerSocket** — passive listen, waits for a message.
    - **Socket** — active connection to a **ServerSocket**, initiates conversation.
    - Sockets provide an **InputStream** and an **OutputStream** at each end of the socket — two-way communication mechanism.
    - Reliable transport — will deliver message or return an exception indicating message no deliverable.
    - Java interface very object-oriented, allows transport of objects, always reliable transport (has a different class for non-reliable transport).
    - C interface uses an untyped byte stream. Only **socket** (no **ServerSocket** equivalent), server uses **bind** after socket creation, client uses **connect** after socket creation.

**RPC and Rendezvous**, Chapter 8, notes: 8-Chapter8.pdf

- RPC has semantics similar to language mechanism.
  - Arguments are automatically marshalled and unmarshalled.
  - Stubs are often involved.
  - Each RPC call creates a thread on the server side. Can result in multiple threads running on Server side.
- Rendezvous.
  - Similar to RPC, but:
    - No new thread is created; thread already exists.
    - “Server” is waiting on (or periodically checking) a receive and handles the incoming call like a message.
    - Arguments are passed — marshalling/unmarshalling like RPC.
- Client/Server:
  - If Server creates a new thread (or forks a new process) to handle the request, then RPC.
  - If Server handles request itself, then this is Rendezvous.
- Sockets in C or Java can be used to re-create RPC and Rendezvous. The one difference is the lack of automatic marshalling/unmarshalling of the arguments.

**RPC and Rendezvous**, (continued):

- Java RMI (Remote Method Invocation).
  - **\_stub** and **\_skel** files on the Client and Server sides (respectively).
    - Handles the marshalling/unmarshalling, sending, receiving.
  - **Naming** and **rmiregistry** handle the binding of the object and its lookup by others.
  - See the **TellAllServer** example.
- Nested RPC's
  - See the File Server and Readers/Writers examples.
  - RPC to remote server, which then does a Rendezvous internally for synchronization.
  - RPC to remote server which then does an RPC/Rendezvous to other servers.
- MPD: **in ... ni**
  - Rendezvous mechanism.
  - “Guarded” statements, **st** and **by** clauses, used to filter incoming messages.
  - **else** clause allows polling.
  - Supports marshalling/unmarshalling of arguments.

**Previous Topics to look over:**

- Atomic actions and Busy-waiting, 2-ProcessesSynchronization.pdf.
  - `< s >` and `< await (B) s >` constructs for atomic actions.
- Critical Section problem, 3-LocksBarriers.pdf.
  - Spin locks.
  - Test and Set.
  - Fairness, Liveness, Starvation.
- Semaphores, 4-Chapter4.pdf.
  - Readers-Writers problem and solution. Note especially “passing the baton”.
- Barriers, 4-Chapter4.pdf.
  - Symmetric, re-usable.
  - Butterfly barrier — “tournament play”, by’s when N not a power of 2.
  - Dissemination barrier — more efficient when N is not a power of 2, easier to implement.