Lecture - IV
- Processes & Threads

Tevfik Koşar

Louisiana State University
January 25th, 2007

Roadmap

• Processes
  - Process Termination
  - Producer-Consumer Problem
  - Inter-process Communication

• Threads
  - Threads vs Processes
  - Multi-threading Models
  - Threading Issues
Process Termination

- Process executes last statement and asks the operating system to delete it (**exit**)
  - Output data from child to parent (**wait**)
  - Process’ resources are deallocated by operating system
- Parent may terminate execution of children processes (**abort**)
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - If parent is exiting
    - Some operating system do not allow child to continue if its parent terminates
      - All children terminated - *cascading termination*

Cooperating Processes

- **Independent** process cannot affect or be affected by the execution of another process
- **Cooperating** process can affect or be affected by the execution of another process
- Advantages of process cooperation
  - Information sharing
  - Computation speed-up
  - Modularity
  - Convenience
Producer-Consumer Problem

- Paradigm for cooperating processes, producer process produces information that is consumed by a consumer process
  - unbounded-buffer places no practical limit on the size of the buffer
  - bounded-buffer assumes that there is a fixed buffer size

Bounded-Buffer - Shared-Memory Solution

- Shared data
  ```
  #define BUFFER_SIZE 10
  typedef struct {
      ...
  } item;
  
  item buffer[BUFFER_SIZE];
  int in = 0;
  int out = 0;
  ```
- Solution is correct, but can only use BUFFER_SIZE-1 elements
Bounded-Buffer - Insert() Method

while (true) {
    /* Produce an item */
    while (((in = (in + 1) % BUFFER SIZE count) == out) 
    ; /* do nothing -- no free buffers */
    buffer[in] = item;
    in = (in + 1) % BUFFER SIZE;
}

Bounded Buffer - Remove() Method

while (true) {
    while (in == out)
    ; // do nothing -- nothing to consume

    // remove an item from the buffer
    item = buffer[out];
    out = (out + 1) % BUFFER SIZE;
    return item;
}
Interprocess Communication (IPC)

- Mechanism for processes to communicate and to synchronize their actions
- **Shared Memory**: by using the same address space and shared variables
- **Message Passing**: processes communicate with each other without resorting to shared variables
- Message Passing facility provides two operations:
  - `send(message)` - message size fixed or variable
  - `receive(message)`
- If \( P \) and \( Q \) wish to communicate, they need to:
  - establish a *communication link* between them
  - exchange messages via `send/receive`

Communications Models

(a) Message Passing

(b) Shared Memory
**Message Passing - direct communication**

- Processes must name each other explicitly:
  - `send(P, message)` - send a message to process P
  - `receive(Q, message)` - receive a message from process Q

- Properties of communication link
  - Links are established automatically
  - A link is associated with exactly one pair of communicating processes
  - Between each pair there exists exactly one link
  - The link may be unidirectional, but is usually bi-directional

- Symmetrical vs Asymmetrical direct communication
  - `send(P, message)` - send a message to process P
  - `receive(id, message)` - receive a message from any process

- Disadvantage of both: limited modularity, hardcoded

**Message Passing - indirect communication**

- Messages are directed and received from mailboxes (also referred to as ports)
  - Each mailbox has a unique id
  - Processes can communicate only if they share a mailbox

- Primitives are defined as:
  - `send(A, message)` - send a message to mailbox A
  - `receive(A, message)` - receive a message from mailbox A
Indirect Communication (cont.)

- Operations
  - create a new mailbox
  - send and receive messages through mailbox
  - destroy a mailbox

- Properties of communication link
  - Link established only if processes share a common mailbox
  - A link may be associated with many processes
  - Each pair of processes may share several communication links
  - Link may be unidirectional or bi-directional

Indirect Communication (cont.)

- Mailbox sharing
  - P₁, P₂, and P₃ share mailbox A
  - P₁, sends; P₂ and P₃ receive
  - Who gets the message?

- Solutions
  - Allow a link to be associated with at most two processes
  - Allow only one process at a time to execute a receive operation
  - Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was.
Synchronization

- Message passing may be either blocking or non-blocking
- **Blocking** is considered **synchronous**
  - **Blocking send** has the sender block until the message is received
  - **Blocking receive** has the receiver block until a message is available
- **Non-blocking** is considered **asynchronous**
  - **Non-blocking** send has the sender send the message and continue
  - **Non-blocking** receive has the receiver receive a valid message or null

Buffering

- Queue of messages attached to the link; implemented in one of three ways
  1. Zero capacity - 0 messages
     Sender must wait for receiver (rendezvous)
  2. Bounded capacity - finite length of \( n \) messages
     Sender must wait if link full
  3. Unbounded capacity - infinite length
     Sender never waits
Motivation

- In certain cases, a single application may need to run several tasks at the same time
  - Create a new process for each task
    - Time consuming
  - Use a single process with multiple threads
Single and Multithreaded Processes

Parent process

Child processes

Process Spawning:
Process creation involves the following four main actions:
- setting up the process control block,
- allocation of an address space and
- loading the program into the allocated address space and
- passing on the process control block to the scheduler
Multi-thread model

Thread Spawning:
• Threads are created within and belonging to processes
• All the threads created within one process share the resources of the
  process including the address space
• Scheduling is performed on a per-thread basis.
• The thread model is a finer grain scheduling model than the process
  model
• Threads have a similar lifecycle as the processes and will be managed
  mainly in the same way as processes are

Threads vs Processes

• Heavyweight Process = Process
• Lightweight Process = Thread

Advantages (Thread vs. Process):
• Much quicker to create a thread than a process
• Much quicker to switch between threads than to switch between processes
• Threads share data easily

Disadvantages (Thread vs. Process):
• Processes are more flexible
  - They don’t have to run on the same processor
• No security between threads: One thread can stomp on another thread’s
  data
• For threads which are supported by user thread package instead of the
  kernel:
  - If one thread blocks, all threads in task block.
**Different Multi-threading Models**

- Many-to-One
- One-to-One
- Many-to-Many

**Many-to-One Model**

- Several user-level threads mapped to single kernel thread
- Thread management in user space → efficient
- If a thread blocks, entire process blocks
- One thread can access the kernel at a time → limits parallelism
- Examples:
  - Solaris Green Threads
  - GNU Portable Threads
One-to-One Model

- Each user-level thread maps to a kernel thread
- A blocking thread does not block other threads
- Multiple threads can access kernel concurrently → increased parallelism
- Drawback: Creating a user level thread requires creating a kernel level thread → increased overhead and limited number of threads
- Examples: Windows NT/XP/2000, Linux, Solaris 9 and later

Many-to-Many Model

- Allows many user level threads to be mapped to a smaller number of kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Increased parallelism as well as efficiency
- Solaris prior to version 9
- Windows NT/2000 with the ThreadFiber package
Two-level Model

- Similar to M:M, except that it allows a user thread to be **bound** to kernel thread
- Examples: IRIX, HP-UX, Tru64 UNIX, Solaris 8 and earlier

Threading Issues

- Semantics of `fork()` and `exec()` system calls
- Thread cancellation
- Signal handling
- Thread pools
- Thread specific data
Semantics of fork() and exec()

- Semantics of fork() and exec() system calls change in a multithreaded program
  - Eg. if one thread in a multithreaded program calls fork()
    - Should the new process duplicate all threads?
    - Or should it be single-threaded?
  - Some UNIX systems implement two versions of fork()

- If a thread executes exec() system call
  - Entire process will be replaced, including all threads

Thread Cancellation

- Terminating a thread before it has finished
  - If one thread finishes the searching a database, others may be terminated
  - If user presses a button on a web browser, web page can be stopped from loading further

- Two approaches to cancel the target thread
  - Asynchronous cancellation terminates the target thread immediately
  - Deferred cancellation allows the target thread to periodically check if it should be cancelled
    - More controlled and safe
Signal Handling

- Signals are used in UNIX systems to notify a process that a particular event has occurred.
- All signals follow this pattern:
  1. Signal is generated by particular event
  2. Signal is delivered to a process
  3. Once delivered, a signal must be handled
- In multithreaded systems, there are 4 options:
  - Deliver the signal to the thread to which the signal applies
  - Deliver the signal to every thread in the process
  - Deliver the signal to certain threads in the process
  - Assign a specific thread to receive all signals for the process

Thread Pools

- Threads come with some overhead as well.
- Unlimited threads can exhaust system resources, such as CPU or memory.
- Create a number of threads at process startup) and put them in a pool, where they await work.
- When a server receives a request, it awakens a thread from this pool.
- Advantages:
  - Usually faster to service a request with an existing thread than create a new thread
  - Allows the number of threads in the application(s) to be bound to the size of the pool
- Number of threads in the pool can be setup according to:
  - Number of CPUs, memory, expected number of concurrent requests
Thread Specific Data

- Threads belonging to the same process share the data of the process
- In some cases, each thread needs to have its own copy of data → thread specific
- Useful when you do not have control over the thread creation process (i.e., when using a thread pool)

Any Questions?

Hmm..
Reading Assignment

• Read chapter 4 from Silberschatz.

Acknowledgements