EDIT 01-11-15 - Added additional answer for problem 1. Process Creation

---

1. Process Creation

Problem

How is a new process created?

Solution

- Via fork
  - If we want to create a process that is an exact replica of the calling process
- Via fork followed by exec
  - If we want to create a process that is not an exact replica of the calling process
- Via fork or fork followed by exec
  - Either of the above two options

Relevant Sections

- P2L1: Processes and Process Management
  - Process Life Cycle: Creation

2. Multi-Threading and 1 CPU

Problem

Is there a benefit of multithreading on 1 CPU?

Solution

Yes. The main reason is to hide the latency associated with code that blocks processing (such as a disk I/O request).

Relevant Sections

- P2L2: Threads and Concurrency
  - Benefits of Multithreading: Single CPU

3. Critical Section

Problem

In the (pseudo) code segments for the producer code and consumer code, mark and explain all the lines where there are errors.

Solution

- Producer code
  - Line 3: uses “if” instead of “while”
  - Line 4: condition_wait doesn’t specify a mutex
  - Line 7: since only 1 item is added, no need to broadcast, should signal instead

    - Line 8: missing the mutex_unlock
- Consumer code
    - Line 3: uses “if” instead of “while”
    - Line 4: condition_wait doesn’t specify a mutex
    - Line 7: condition_signal signals the wrong variable, should be signaling not_full
    - Line 8: missing the mutex_unlock operation

Relevant Sections

- P2L2: Threads and Concurrency (the entire lesson is useful for this problem)

4. Calendar Critical Section

Problem

A shared calendar supports three types of operations for reservations:

1. read
2. cancel
3. enter

Requests for cancellations should have priority above reads, who in turn have priority over new updates. In pseudocode, write the critical section enter/exit code for the **read** operation.

Solution

We have not posted a solution for this problem. Instead, since there are many possible solutions, we encourage you to post and share your solutions on the class forum.

Relevant Sections

- P2L2: Threads and Concurrency (the entire lesson is useful for this problem)

5. Signals

Problem

If the kernel cannot see user-level signal masks, then how is a signal delivered to a user-level thread (where the signal can be handled)?

Solution

Recall that all signals are intercepted by a user-level threading library handler, and the user-level threading library installs a handler. This handler determines which user-level thread, if any, the signal be delivered to, and then it takes the appropriate steps to deliver the signal.

Note: If all user-level threads have the signal mask disabled and the kernel-level signal mask is updated, then and the signal remains pending to the process.

Relevant Sections

- P2L4: Thread Design Considerations
    - All morsels including and after the Interrupts and Signals Intro

6. Solaris Papers

Problem

The implementation of Solaris threads described in the paper "Beyond Multiprocessing: Multithreading the Sun OS Kernel", describes four key data structures used by the OS to support threads.

For each of these data structures, list at least two elements they must contain:

1. Process
2. LWP
3. Kernel-threads
4. CPU

Solution

The video Kernel Level Structures in Solaris 2.0 in **Relevant Sections** contains many possible solutions.

Relevant Sections

- P2L4: Thread Design Considerations
    - Kernel Level Structures in Solaris 2.0

7. Pipeline Model

Problem

An image web server has three stages with average execution times as follows:

- Stage 1: read and parse request (10ms)
- Stage 2: read and process image (30ms)
- Stage 3: send image (20ms)

You have been asked to build a multi-threaded implementation of this server using the **pipeline model**. Using a **pipeline model**, answer the following questions:

1. How many threads will you allocate to each pipeline stage?
2. What is the expected execution time for 100 requests (in sec)?
3. What is the average throughput of this system (in req/sec)? Assume there are infinite processing resources (CPU's, memory, etc.).

Solution

1. Threads should be allocated as follows:
    - Stage 1 should have 1 thread
        - This 1 thread will parse a new request every 10ms
    - Stage 2 should have 3 threads
        - The requests parsed by Stage 1 get passed to the threads in Stage 2. Each thread picks up a request and needs 30ms to process the image. Hence, we need 3 threads in order to pick up a new request as soon as Stage 1 passes it.
    - Stage 3 should have 2 threads.
        - This is because Stage 2 will process an image and pass a request every 10ms (once the pipeline is filled). In this way, each Stage 3 thread will pick up a request and send an image in 20ms. Once the pipeline is filled, Stage 3 will be able to pick up a request and send the image every 10ms.
2. The first request will take 60ms. The last stage will continue delivering the remaining 99 requests at 10ms intervals. So, the total is 60 + (99 * 10ms) = 1050ms = 1.05s
3. 100 req / 1.05 sec = 95.2 req/s

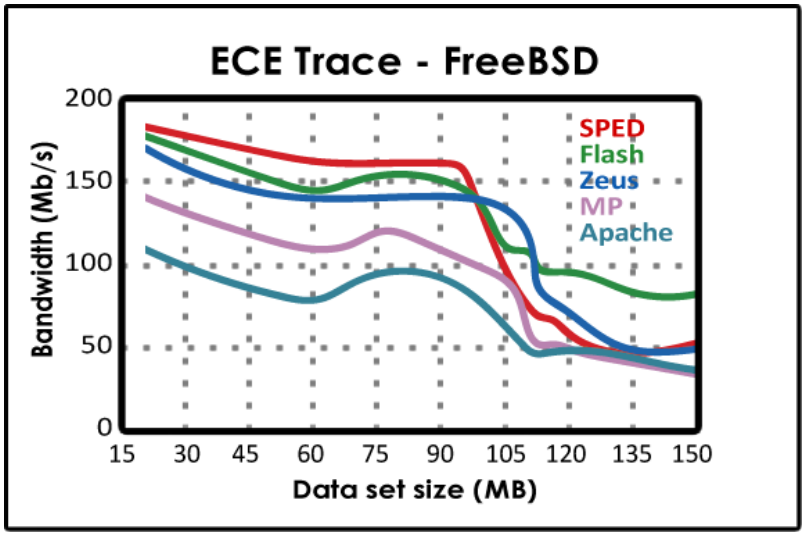
Relevant Sections

- P2L2: Threads and Concurrency
    - Pipeline Pattern
    - Multithreading Patterns Quiz

8. Performance Observations

Problem

Here is a graph from the paper "Flash: An Efficient and Portable Web Server", that compares the performance of Flash with other web servers.



For data sets **where the data set size is less than 100MB** why does...

1. Flash perform worse than SPED?
2. Flash perform better than MP?

Solution

1. In both cases the dataset will likely fit in cache, but Flash incurs an overhead on each request because Flash must first check for cache residency. In the SPED model, this check is not performed.
2. When data is present in the cache, there is no need for slow disk I/O operations. Adding threads or processes just adds context switching overheads, but there is no benefit of "hiding I/O latency".

Relevant Sections

- P2L5: Thread Performance Considerations (the entire lesson is useful for this problem)