Threads and Concurrency

Notes Overview

- What are threads?
- How are threads different from processes?
- What data structures are used to implement and manage threads?

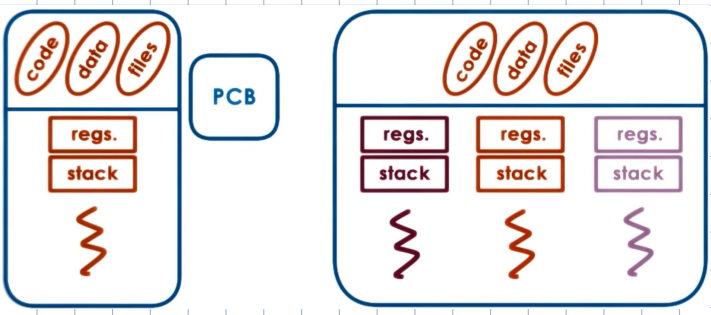
Toyshop Metaphor

- A thread is like a worker in a toy shop
  - Is an active entity
    - executing a unit of a toy order
  - works simultaneously with others
    - many workers completing toy orders
  - requires coordination
    - sharing of tools, parts, and workstations

What about threads

- is an active entity
  - executing a unit of a process
- works simultaneously with others
  - many threads executing
- requires coordination
  - sharing of I/O devices, CPUs, and memory...

Process vs. Thread



Why are threads useful?

Resource Efficiency

- Threads share the same memory space and resources within a process, making them more lightweight than full processes
- Thread creation and context switching between threads is significantly faster than with processes

Parallel Processing

- Multiple threads can execute different parts of a program simultaneously on different CPU cores
- Computationally intensive tasks can be split among threads to improve performance

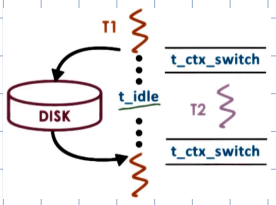
Responsiveness

- Applications can remain responsive by running intensive operations in background threads
- User interface threads can continue handling input while other threads process data

Resource Sharing

- threads within the same process can easily share data and communicate
- Shared memory access allows efficient data exchange between threads

Are threads useful on a single CPU? or when(# of threads) > (# of CPUs)?



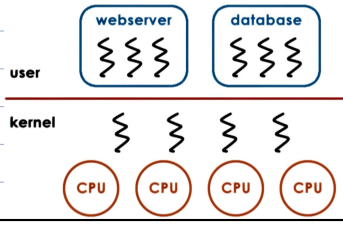
- A context switch is beneficial when t_idle > 2 * t_ctx_switch
- This means the idle time should be more than the context switch time to make threading worthwhile
- The system does this to hide latency during I/O operations

- T1 and T2 are interacting with disk I/O operations
- t_idle represents the idle waiting time during disk operations
- t_ctx_switch represents the time needed for context switching

Practical Application

- The diagram demonstrates how threading can hide latency during disk I/O operations:
  - while one thread (T1) is waiting for disk I/O
  - Another thread (T2) can execute
- The design pattern is particularly useful for I/O bound applications, even on a single CPU system, as it maximizes CPU utilization during I/O wait times

Benefits to applications and OS code



multithreaded applications

multithreaded OS kernel
- threads working on behalf of apps
- OS-level services like daemons or drivers

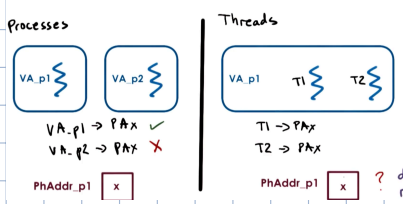
What do we need to support threads?

- thread data structure
  - identify threads, keep track of resource usage...
- mechanisms to create and manage threads
- mechanisms to safely coordinate among threads running concurrently in the same address space

Synchronization Mechanisms

- mutual exclusion
  - exclusive access to only one thread at a time
  - mutex
- waiting on other threads
  - specific condition before proceeding
  - condition variable
- waking up other threads from wait state

Threads and Concurrency



- Processes (VA_p1 and VA_p2) maintain complete memory isolation, where VA_1 can access its physical address x (PhAddr_p1) as indicated by the check mark, while VA_p2 is prevented from accessing it as shown by the cross (X)

Thread Sharing

- Multiple threads exist within the same process VA_p1
- Both threads share the same address space and can access the same physical memory location (PhAddr_p1)
- The shared access can lead to potential data races, indicated by the ? mark

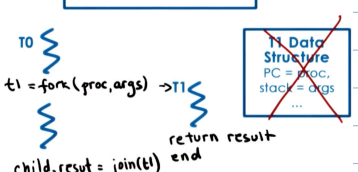
Threads and Thread Creation

Thread type
- thread data structure

Thread type
thread ID, PC, SP, registers, stack, attributes

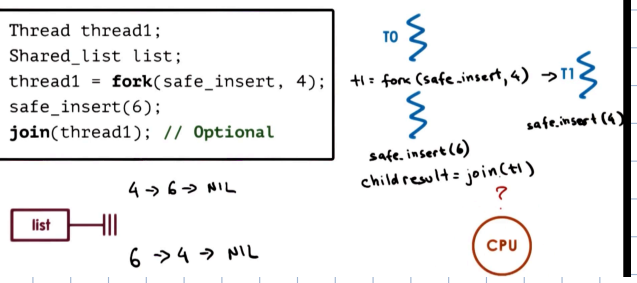
Fork (proc, args)
- create a thread
- not UNIX fork

Join (thread)
- terminate a thread



Thread Creation Example

```
Thread thread1;
Shared_list list;
thread1 = fork(safe_insert, 4);
safe_insert(6);
join(thread1); // Optional
```



Mutual Exclusion

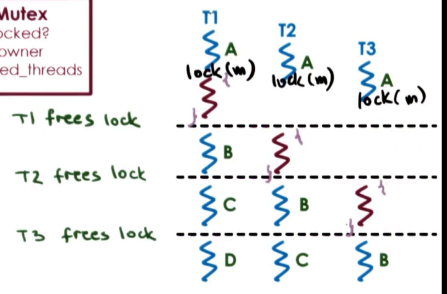
Mutex

Lock (mutex) {

Critical Section

}

Mutex
locked?
owner
blocked_threads



Mutual Exclusion

Mutex

Lock (mutex) {

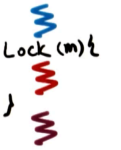
Critical Section

}

Lock (m)/
Unlock (m)

Birrell's Lock API

```
Lock(m) {
    // critical section
} // unlock;
```



Common Thread API

```
lock(m);

// critical section

unlock(m);
```

Mutex Implementation

- A mutex is a synchronization primitive that forces exclusive access to a critical section. At its most fundamental level, a mutex is implemented as:
  - a boolean in memory indicating lock status
  - two atomic operations: lock/unlock
  - a queue for waiting threads

How a Mutex Works

When a thread attempts to acquire a mutex:

1. the thread checks if the mutex is available through an atomic operation
2. If available, the thread:
   - Sets the mutex to a locked state
   - Becomes the mutex owner
   - Enters the critical section
3. If unavailable, the thread:
   - Gets blocked
   - Enters a wait queue
   - Yields execution until the mutex is released

Unlock Operation

When the owner thread completes its critical section:

1. the mutex is set to an unlocked state
2. one waiting thread (if any) is awakened
3. The awakened thread becomes the new owner

Condition Variable

- Should be used in conjunction with mutexes to control the behavior of concurrent threads
- A condition variable is a synchronization primitive that enables threads to wait until a specific condition occurs. They are used in conjuction with mutexes to handle complex synchronization scenarios where threads need to coordinate based on program state

Condition Variable Structure

- wait queue for blocked threads
- associated mutex for protected shared state
- atomic operations for signaling and waiting

Signal Operations

- Signal (notify-one):
  - Wakes up a single thread from the wait queue
  - Used when only one thread needs to be notified
- Broadcast (notify-all):
  - Wakes up all waiting threads
  - Used when all waiting threads need to be notified

Usecases

Producer-consumer Pattern

- Producers signal when data is available
- Consumers wait for data to become available
- prevents busy-waiting and ensures efficient resource usage

Condition Variable API

Condition type

Wait (mutex, cond)
- mutex is automatically released & re-acquired on wait

Signal (cond)
- notify only one thread waiting on condition

Broadcast (cond)
- notify all waiting threads

Condition Variable
mutex ref
waiting threads
...

```
Wait(mutex, cond) {
    // atomically release mutex
    // and go on wait queue

    // ... wait ... wait ... wait ...

    // remove from queue
    // re-acquire mutex
    // exit the wait operation
}
```

Readers / Writer Problem

Readers

0 or more

if (read_counter == ø) and (write_counter == ø)
→ R ok, W ok

if (read_counter > ø)
→ R ok

if (writer_count == 1)
→ R no, W no

Writer

0 or 1

State of shared file / resource:
- free : resource_counter = 0
- reading: resource_counter > ø
- writing: resource_counter = -1

Readers / Writer Problem

Readers

0 or more

if (read_counter == ø) and (write_counter == ø)
→ R ok, W ok

if (read_counter > ø)
→ R ok

if (writer_count == 1)
→ R no, W no

Writer

0 or 1

State of shared file / resource:
- free : resource_counter = 0
- reading: resource_counter > ø
- writing: resource_counter = -1

The core challenge

The main challenge is ensuring that:

- **Multiple readers can coexist**: If only readers are active, they should be allowed to access resources concurrently to maximize performance
- **Writers need exclusive access**: when a writer is active, no other reader or writer should access the resource
- **Balancing priorities**: A naive implementation might lead to writer or reader starvation

Reader / Writer Example ①

```
Mutex counter_mutex; Condition read_phase, write_phase; int resource_counter = 0;
```

```
// READERS
Lock(counter_mutex) {
  while(resource_counter == -1)
    Wait(counter_mutex, read_phase);
  resource_counter++;
} // unlock;
// ... read data ...
Lock(counter_mutex) {
  resource_counter--;
  if(readers == 0)
    Signal(write_phase);
} // unlock;
```

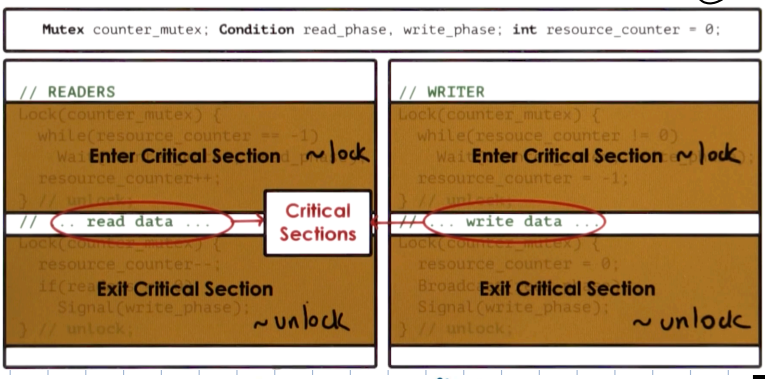
```
// WRITER
Lock(counter_mutex) {
  while(resouce_counter != 0)
    Wait(counter_mutex, write_phase);
  resource_counter = -1;
} // unlock;
// ... write data ...
Lock(counter_mutex) {
  resource_counter = 0;
  Broadcast(read_phase);
  Signal(write_phase);
} // unlock;
```

Critical Sections in Readers / Writer Example

Typical Critical Section Structure ②

```
Lock(mutex) {
  while(!predicate_indicating_access_ok)
    wait (mutex, cond_var)
  update state => update predicate
  signal and/or broadcast
    (cond_var with correct waiting threads)
} // unlock;
```

③



Critical Section Structure with Proxy Variable

```
// ENTER CRITICAL SECTION
perform critical operation (read/write shared file)
// EXIT CRITICAL SECTION
```

```
// ENTER CRITICAL SECTION
Lock(mutex) {
  while(!predicate_for_access)
    wait(mutex, cond_var)
  update predicate
} // unlock;
```

```
// EXIT CRITICAL SECTION
Lock(mutex) {
  update predicate;
  signal/broadcast(cond_var)
} // unlock;
```

Avoiding Common Mistakes

- Keep track of mutex/cond. variables used with a resource
  - e.g., mutex_type m1; // mutex for file1
- Check that you are always (and correctly) using lock & unlock
  - e.g., did you forget to lock / unlock? What about compilers?
- Use a single mutex to access a single resource!

```
Lock(m1) {
//read file1
} // unlock;
```

```
Lock(m2) {
//write file1
} // unlock;
```

=> read and writes allowed to happen concurrently!

Avoiding Common Mistakes

- Check that you are signaling correct condition
- Check that you are not using signal when broadcast is needed
  - signal: only 1 thread will proceed ... remaining threads will continue to wait ... possibly indefinitely!!!
- Ask yourself: do you need priority guarantees?
  - thread execution order not controlled by signals to condition variables!

Spurious Wake-Ups

If (unlock after broadcast/signal) => no other thread can get lock!

```
// WRITER
Lock(counter_mutex) {
  resource_counter = 0;
  Broadcast(read_phase);
  Signal(write_phase);
} // unlock;
```

```
// READERS
// elsewhere in the code ...
Wait(counter_mutex,
write/read_phase);
```

spurious wake-ups == when we wake threads up knowing they may not be able to proceed.

=> spurious wake-up

A spurious wake-up happens when a thread that's waiting for a specific event wakes up without any clear reason. Because these unexpected wakeups can occur, it's always important to check if the condition you're waiting for is actually met by proceeding. This is why you usually see a loop that re-checks the condition after waking up, ensuring that the thread really continues when it is really supposed to

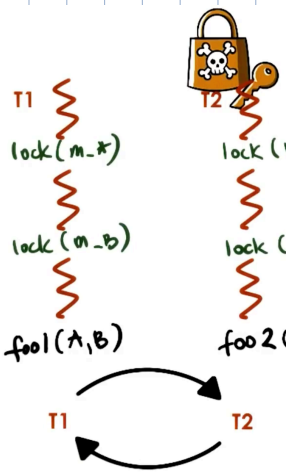
more on how to avoid deadlocks below ↓

Deadlocks

Definition:

Two or more competing threads are waiting on each other to complete, but none of them ever do.

Deadlocks



How to avoid this?

- Maintain lock order
  - first m_A
  - then m_B

⊕ Will prevent cycles in wait graph

Deadlocks

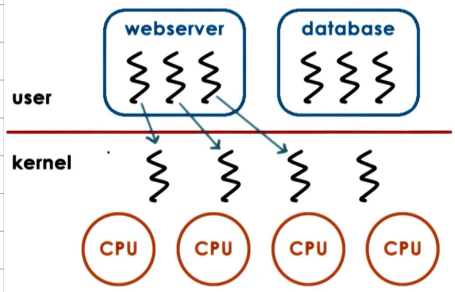
In summary
A cycle in the wait graph is necessary and sufficient for a deadlock to occur
(edges from thread waiting on a resource to thread owning a resource)

What can we do about it?
- Deadlock prevention EXPENSIVE
- Deadlock detection & recovery ROLLBACK
- Apply the Ostrich Algorith ... DO NOTHING!

=> if all else fails ... just REBOOT

Kernel vs. User-level Threads

One-to-One Model:

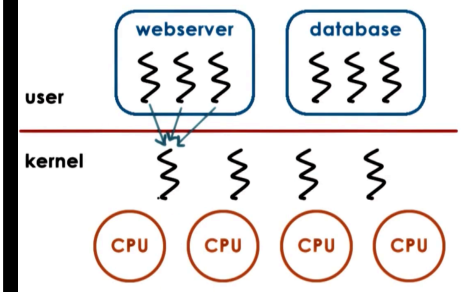


⊕ + OS sees/understands threads, synchronization, blocking ...

⊖ - must go to OS for all operations (may be expensive)
- OS may have limits on policies, thread #
- portability

Kernel vs. User-level Threads

Many-to-One Model:

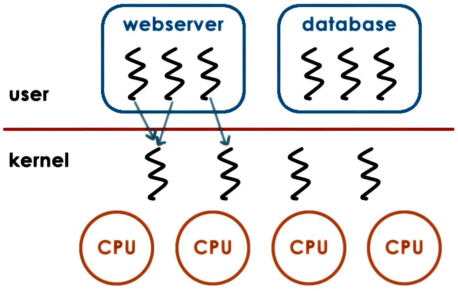


⊕ + totally portable, doesn't depend on OS limits and policies

⊖ - OS has no insights into application needs
- OS may block entire process if one user-level thread blocks on I/O

Kernel vs. User-level Threads

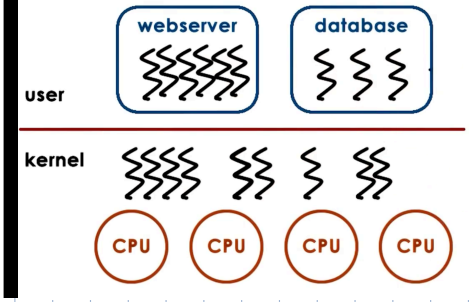
Many-to-Many Model:



⊕ + can be best of both worlds
+ can have bound or unbound threads

⊖ - requires coordination between user- and kernel-level thread managers

Kernel vs. User-level Threads

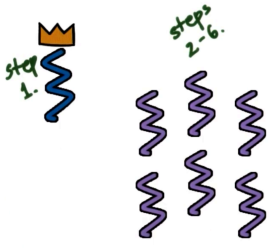


Process Scope:
User-level library manages threads within a single process

System Scope:
System-wide thread management by OS-level thread managers (e.g., CPU scheduler)

Boss-Workers Pattern

①



Boss-Workers:
- boss: assigns work to workers
- worker: performs entire task

Throughput of the system limited by boss thread => must keep boss efficient

Throughput = $1 / boss\_time\_per\_order$

How does the boss thread pass work to the worker threads?

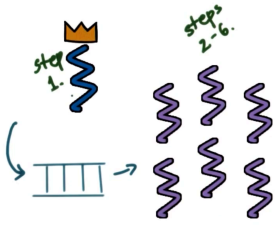
- directly signaling specific workers

⊕ + workers do not need to synchronize

⊖ - boss must track what each worker is doing
- throughput will go down!

Boss-Workers Pattern

②



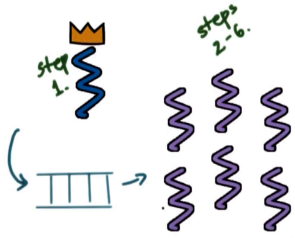
Boss-Workers:
- boss: assigns work to workers
- worker: performs entire task

Boss assigns work by:
- placing work in producer/consumer queue

⊕ - boss doesn't need to know details about workers

⊖ - queue synchronization

Boss - Workers Pattern

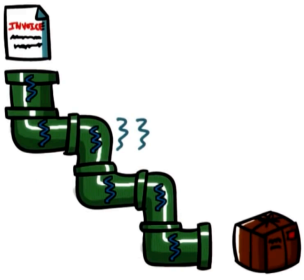


Boss - Workers:
- boss: assigns work to workers
- worker: performs entire task
- boss-workers communicate via producer/consumer queue
- worker pool: static or dynamic

⊕ + simplicity

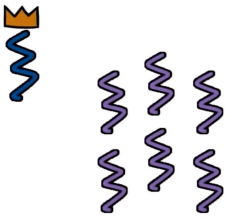
⊖ - thread pool management
- locality

Pipeline Pattern



Pipeline:
- threads assigned one subtask in the system
- entire tasks == pipeline of threads
- multiple tasks concurrently in the system, in different pipeline stages
- throughput == weakest link => pipeline stage == thread pool
- shared-buffer based communication b/w stages

Boss - Workers Pattern



Boss - Workers Variants:
- all workers created equal

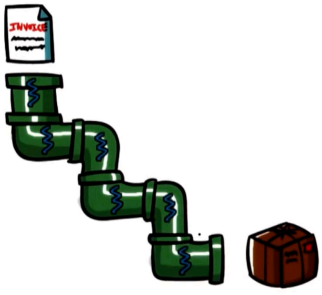
vs.

- workers specialized for certain tasks

⊕ - better locality; QoS management

⊖ - load balancing

Pipeline Pattern



Pipeline:
- sequence of stages
- stage == subtask
- each stage == thread pool
- buffer-based communication

⊕ + specialization and locality

⊖ - balancing and synchronization overheads

Layered Pattern

Layered:
- each layer group of related subtasks
- end-to-end task must pass up and down through all layers

⊕ + specialization
+ less fine-grained than pipeline

⊖ - not suitable for all applications
- synchronization