CS 111 Final Review Session

Winter 2023

Briana Berger and Yashodhar Govil

Slides by Briana Berger, Yashodhar Govil Parthiv Krishna

Key Topics

- Filesystems and Crash Recovery
- Multiprocessing and Pipes
- **Multithreading and Synchronization**
- **Dispatching and Scheduling**
- **Virtual Memory and Paging**
- **Ethics**

⚡ Lightning Recap ⚡
Filesystems n Multiprocessing

*Exam emphasizes knowledge from the later half of the class though; thus this isn’t comprehensive

Unix v6 Filesystem

- Stores inodes on disk together in the inode table for quick access.
- An inode ("index node") is a grouping of data about a single file. It’s stored on disk, but we can read it into memory when the file is open
  - Each Unix v6 inode has space for 8 block numbers
- For "small" files/directories, i_addr stores up to 8 direct block numbers.
- For "large" files/directories, i_addr's up to first seven entries store singly-indirect block numbers, and the eighth entry (if needed) stores a doubly-indirect block number.

3 Approaches to Crash Recovery

1. Consistency check on reboot (fsck)

2. Ordered Writes

3. Write-Ahead Logging (“Journaling”)

3 Approaches to Crash Recovery

1. Consistency check on reboot (fsck)
    a. No filesystem changes, run program on boot to repair whatever we can.
    b. Downsides: Doesn’t prevent information loss & filesystem may still be unusable
2. Ordered Writes

3. Write-Ahead Logging (“Journaling”)

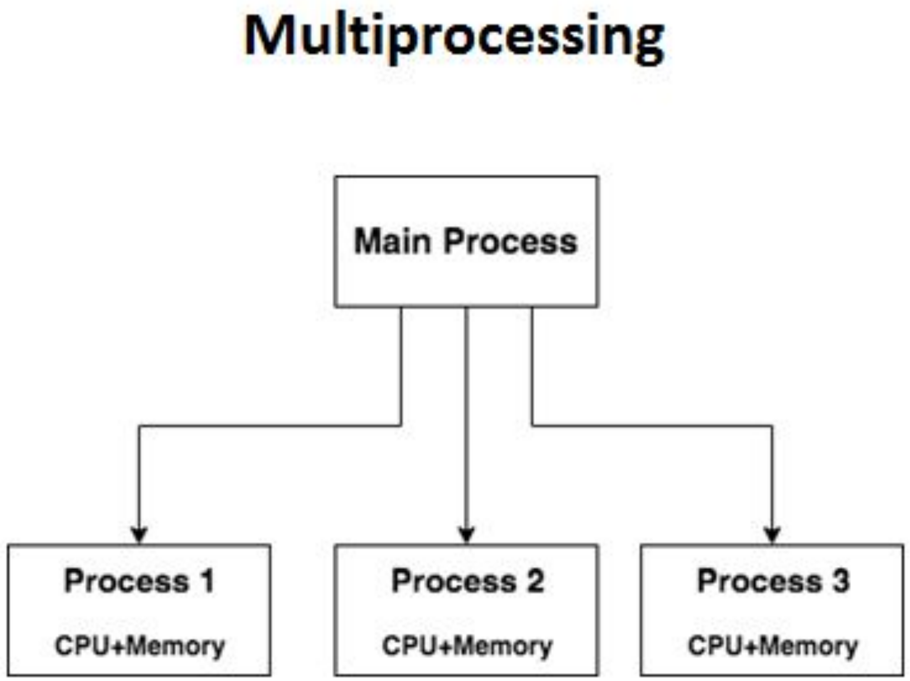
3 Approaches to Crash Recovery

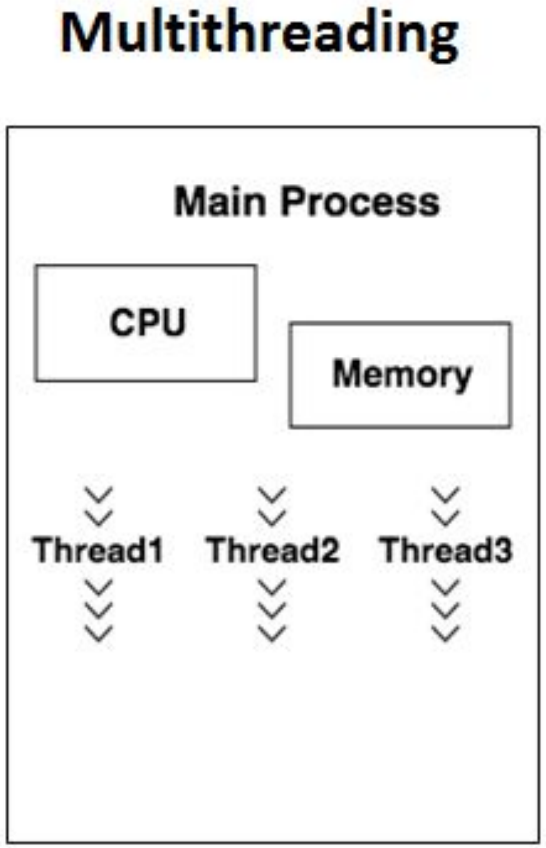
1. Consistency check on reboot (fsck)
    a. No filesystem changes, run program on boot to repair whatever we can.
    b. Downsides: Doesn’t prevent information loss & filesystem may still be unusable
2. Ordered Writes
    a. We could prevent certain inconsistencies by making writes in a particular order.
    b. Downsides: dependency management, leaks data
3. Write-Ahead Logging (“Journaling”)

3 Approaches to Crash Recovery

1. Consistency check on reboot (fsck)
    a. No filesystem changes, run program on boot to repair whatever we can.
    b. Downsides: Doesn’t prevent information loss & filesystem may still be unusable
2. Ordered Writes
    a. We could prevent certain inconsistencies by making writes in a particular order.
    b. Downsides: dependency management, leaks data
3. Write-Ahead Logging (“Journaling”)
    a. log metadata (and optionally file data) operations before doing the operations to create a paper trail we can redo in case of a crash.
    b. must be done synchronously (soln: delay writes), log gets long (soln: checkpoints), multiple logs might need some operations to be atomic (soln: transactions), logs must be idempotent (doing multiple times has same effect as doing once)

Multiprocessing (fork) vs. Multithreading (std::thread)





Concurrency within a single process using threads!

Multiprocessing (fork) vs. Multithreading (std::thread)

- Concurrency within a single process using threads.
- Processes:
  - isolate virtual address spaces (✅: security, 🚩: harder to share)
  - run external programs easily (fork-exec) (✅)
  - harder to coordinate tasks in the same program (🚩)
- Threads:
  - share virtual address space (🚩: security, ✅: easier to share)
  - can't run external programs easily (🚩)
  - easier to coordinate tasks within the same program (✅)

What questions do you have?

Multithreading and Synchronization

The Monitor Pattern: ThreadPipe

ThreadPipe

- Let's implement a class called `ThreadPipe`
- Like a pipe, but between threads instead of processes
- `void put(char c);`
    - Puts a character in the pipe (or blocks if it's full, just like `write` to a pipe)
- `char get();`
    - Gets a character from the pipe (or blocks if it's empty, just like `read` from a pipe)

ThreadPipe: Baseline Implementation

```
class ThreadPipe {
   ThreadPipe() {}
   void put(char c);
   char get();

   char buffer[SIZE];
   int count = 0;
   int nextPut = 0;
   int nextGet = 0;
};
```

```
void ThreadPipe::put(char c) {
     count++;
     buffer[nextPut] = c;
     nextPut++;
     if (nextPut == SIZE) {
          nextPut = 0;
     }
}

char ThreadPipe::get() {
     count--;
     char c = buffer[nextGet];
     nextGet++;
     if (nextGet == SIZE) {
          nextGet = 0;
     }
     return c;
}
```

ThreadPipe: Baseline Implementation

```
class ThreadPipe {
   ThreadPipe() {}
   void put(char c);
   char get();

   char buffer[SIZE];
   int count = 0;
   int nextPut = 0;
   int nextGet = 0;
};
```

```
void ThreadPipe::put(char c) {
     count++;
     buffer[nextPut] = c;
     nextPut++;
     if (nextPut == SIZE) {
          nextPut = 0;
     }
}

char ThreadPipe::get() {
     count--;
     char c = buffer[nextGet];
     nextGet++;
     if (nextGet == SIZE) {
          nextGet = 0;
     }
     return c;
}
```

Are there any race conditions possible? If so, how can we fix it?

ThreadPipe: Locked Implementation

```
class ThreadPipe {
   ThreadPipe() {}
   void put(char c);
   char get();


   std::mutex lock;
   char buffer[SIZE];
   int count = 0;
   int nextPut = 0;
   int nextGet = 0;
};
```

```
void ThreadPipe::put(char c) {
     lock.lock();
     count++;
     buffer[nextPut] = c;
     nextPut++;
     if (nextPut == SIZE) {
          nextPut = 0;
     }

     lock.unlock();
}

char Pipe::get() {
     lock.lock();
     count--;
     char c = buffer[nextGet];
     nextGet++;
     if (nextGet == SIZE) {
          nextGet = 0;
     }
     lock.unlock();
     return c;
}
```

ThreadPipe: Locked Implementation

```
class ThreadPipe {
   ThreadPipe() {}
   void put(char c);
   char get();


   std::mutex lock;
   char buffer[SIZE];
   int count = 0;
   int nextPut = 0;
   int nextGet = 0;
};
```

What if the ThreadPipe is full/empty?

```
void ThreadPipe::put(char c) {
     lock.lock();
     count++;
     buffer[nextPut] = c;
     nextPut++;
     if (nextPut == SIZE) {
          nextPut = 0;
     }

     lock.unlock();
}

char Pipe::get() {
     lock.lock();
     count--;
     char c = buffer[nextGet];
     nextGet++;
     if (nextGet == SIZE) {
          nextGet = 0;
     }
     lock.unlock();
     return c;
}
```

ThreadPipe: Busywaiting

```
class ThreadPipe {
   ThreadPipe() {}
   void put(char c);
   char get();

   std::mutex lock;
   char buffer[SIZE];
   int count = 0;
   int nextPut = 0;
   int nextGet = 0;
};
```

```
void ThreadPipe::put(char c) {
    lock.lock();
    while (count == SIZE) {
        lock.unlock();
        lock.lock();
    }
    count++;
    buffer[nextPut] = c;
    nextPut++;
    if (nextPut == SIZE) {
        nextPut = 0;
    }

    lock.unlock();
}

char Pipe::get() {
    lock.lock();
    while (count == 0) {
        lock.unlock();
        lock.lock();
    }
    count--;
    char c = buffer[nextGet];
    nextGet++;
    if (nextGet == SIZE) {
        nextGet = 0;
    }
    lock.unlock();
    return c;
}
```

ThreadPipe: Busywaiting

```
class ThreadPipe {
   ThreadPipe() {}
   void put(char c);
   char get();


   std::mutex lock;
   char buffer[SIZE];
   int count = 0;
   int nextPut = 0;
   int nextGet = 0;
};
```

How can we avoid busywaiting?

```
void ThreadPipe::put(char c) {
    lock.lock();
    while (count == SIZE) {
        lock.unlock();
        lock.lock();
    }
    count++;
    buffer[nextPut] = c;
    nextPut++;
    if (nextPut == SIZE) {
        nextPut = 0;
    }

    lock.unlock();
}

char Pipe::get() {
    lock.lock();
    while (count == 0) {
        lock.unlock();
        lock.lock();
    }
    count--;
    char c = buffer[nextGet];
    nextGet++;
    if (nextGet == SIZE) {
        nextGet = 0;
    }
    lock.unlock();
    return c;
}
```

Condition Variables

1. Identify a single kind of event that we need to wait / notify for
2. Ensure there is proper state to check if the event has happened
3. Create a condition variable and share it among all threads either waiting for that event to happen or triggering that event
4. Identify who will notify that this happens, and have them notify via the condition variable
5. Identify who will wait for this to happen, and have them wait via the condition variable

ThreadPipe: Condition Variables

```
class ThreadPipe {
    ThreadPipe() {}
    void put(char c);
    char get();


    std::mutex lock;
    std::condition_variable_any added;
    std::condition_variable_any removed;

    char buffer[SIZE];
    int count = 0;
    int nextPut = 0;
    int nextGet = 0;
};
```

```
void ThreadPipe::put(char c) {
    lock.lock();
    while (count == SIZE) {
        removed.wait(lock);
    }
    count++;
    buffer[nextPut] = c;
    nextPut++;
    if (nextPut == SIZE)
        nextPut = 0;
    if (count == 1)
        added.notify_all();
    lock.unlock();
}
```

```
char Pipe::get() {
    lock.lock();
    while (count == 0) {
        added.wait(lock);
    }
    count--;
    char c = buffer[nextGet];
    nextGet++;
    if (nextGet == SIZE)
        nextGet = 0;

    if (count == SIZE-1)
        removed.notify_all();
    lock.unlock();
    return c;
}
```

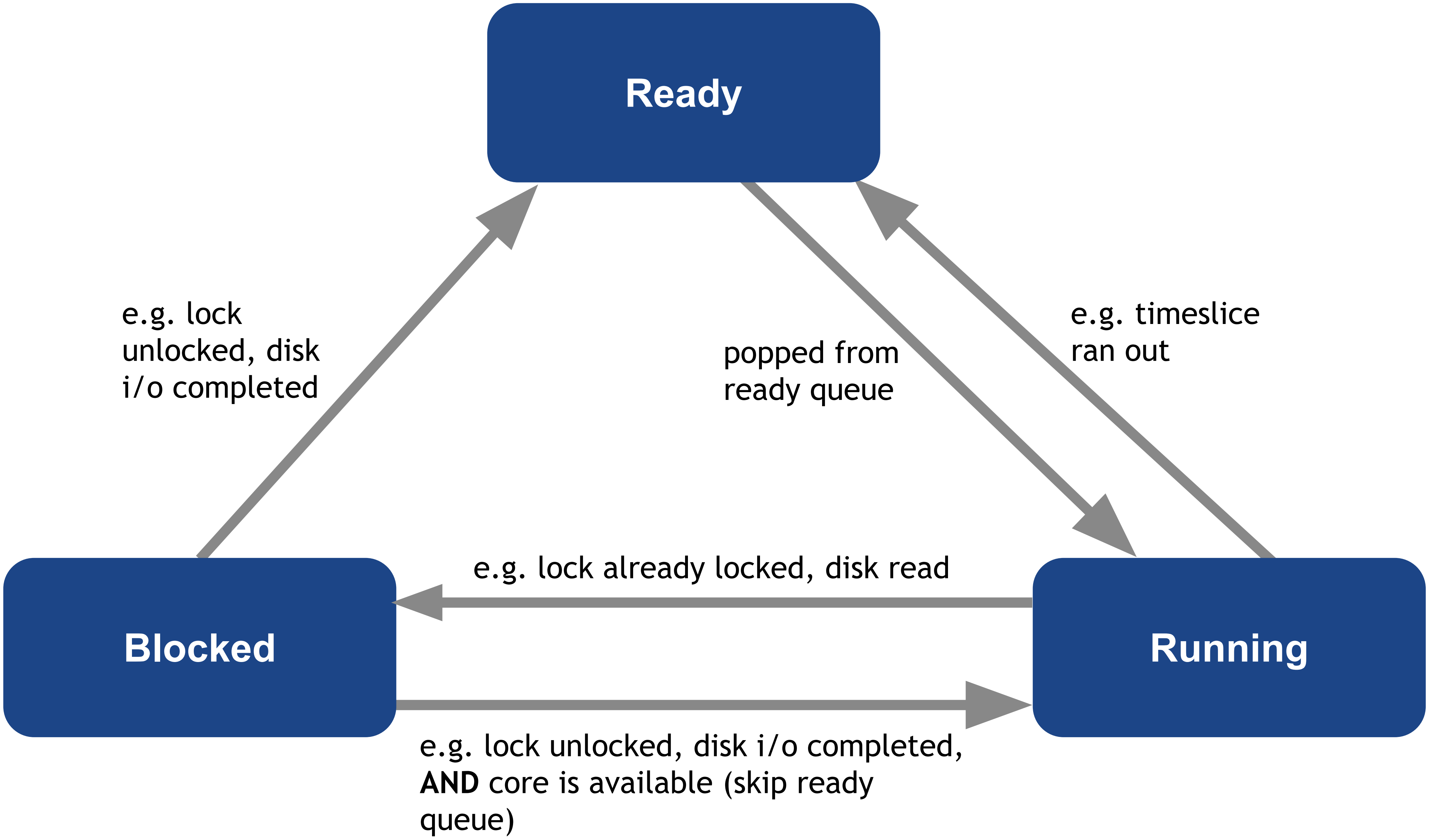
What questions do you have about ThreadPipe?

Dispatching and Scheduling

110 Practice Final 3: Question 4e

e. [2 points] The process scheduler relies on runnable and blocked queues to categorize processes. How exactly does this categorization lead to better CPU utilization?

- Don’t want to run threads that can’t do any useful work right now (blocked).
- Ensures that we only run threads that can do something.

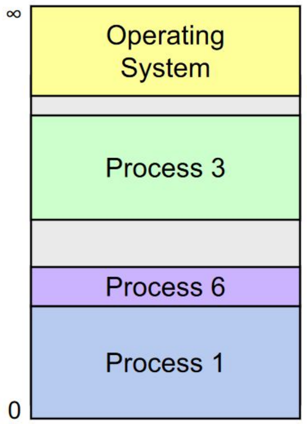

Ready
Blocked
Running
e.g. lock unlocked, disk i/o completed
popped from ready queue
e.g. timeslice ran out
e.g. lock already locked, disk read
e.g. lock unlocked, disk i/o completed, **AND** core is available (skip ready queue)

What questions do you have about Dispatching and Scheduling?

Virtual Memory

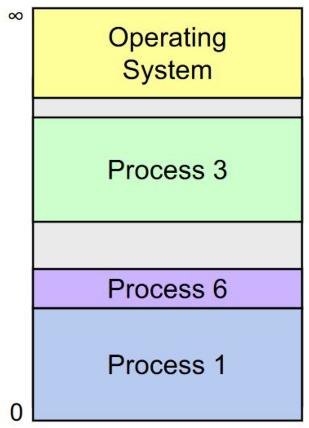
Different Approaches: Pros and Cons

Load Time Relocation



- Pros
- Cons

Load Time Relocation



- Pros
  - Fast once loading is done (no address translation needed)
- Cons
  - Must decide process memory space ahead of time
  - Cannot grow when adjacent regions are used
  - External fragmentation
  - Programs are compiled assuming their memory space starts at 0, so we would need to rewrite the program’s pointers when we load
    - Can’t move the program in memory after loading unless we somehow intercept and update all pointers

Base and Bound

- Pros
- Cons

Base and Bound

- Pros
  - Simple
  - Quick address translation
  - Very little space needed to track info about each process's memory
- Cons
  - All memory allocated to a process has to be contiguous virtual addresses
    - Stack is often far from heap in virtual address space
  - Can only grow upwards

Multiple Segments

- Pros
- Cons

Multiple Segments

- Pros
  - Not as simple as Base + Bound, but still very simple
  - Still pretty quick address translation
  - Still relatively little space needed per-process for VM info
  - Can allocate different discontinuous areas of VM with different protections
    - Code, Data, Stack
- Cons
  - Segments are of different sizes, so we will tend towards external fragmentation
  - Generally, not many segments

Paging

- Pros
- Cons

Paging

- Pros
  - Fixed size pages: no external fragmentation
  - Can dynamically resize memory allocated to a process
  - Can grow in either direction
  - Can assign different permissions to different pages
    - Code, Data, Stack
- Cons
  - Internal fragmentation within pages. You can only get memory in 4KB chunks.
  - Relatively slower/more complicated address translation, especially with multi-level page tables
    - Can be accelerated with dedicated hardware like memory management unit (MMU)

What questions do you have about Virtual Memory?

Ethics

Agency and Trust

- Trusting software is extending agency
- Agential gullibility

Agency and Trust

- Trusting software is extending agency
  - “when we trust, we try to make something a part of our agency, and we are betrayed when our part lets us down. To unquestioningly trust something is to let it in—to attempt to bring it inside one’s practical functioning.”
  - **Example**: glucose monitoring
- Agential gullibility

Agency and Trust

- Trusting software is extending agency
  - “when we trust, we try to make something a part of our agency, and we are betrayed when our part lets us down. To unquestioningly trust something is to let it in—to attempt to bring it inside one’s practical functioning.”
  - **Example**: glucose monitoring
- Agential gullibility
  - Trusting more than warranted
  - Difficult to judge how trust is warranted given how quickly software changes, hard to inspect
  - **Example**: glucose monitoring issues w/ Android update

Three Paths to Trust: Trust by —

1. Assumption:

2. Inference:

3. Substitution:

Three Paths to Trust: Trust by —

1. Assumption: trust absent any cluses to warrant it.
   a. E.g. using unknown third party library b/c deadline nearing
2. Inference:

3. Substitution:

Three Paths to Trust: Trust by —

1. Assumption: trust absent any cluses to warrant it.
   a. E.g. using unknown third party library b/c deadline nearing
2. Inference: reputation is based on past performance
   a. Log of past actions
   b. Trust in brands
   c. Trust in prior versions of software
3. Substitution:

Three Paths to Trust: Trust by —

1. Assumption: trust absent any cluses to warrant it.
   a. E.g. using unknown third party library b/c deadline nearing
2. Inference: reputation is based on past performance
   a. Log of past actions
   b. Trust in brands
   c. Trust in prior versions of software
3. Substitution: structural arrangements that partly substitute need for trust
   a. Often involves separation of code, responsibilities
   b. E.g. user permissions of file system, separating self-driving functionality of car from infotainment

Trust Examples

- Meltdown - hardware-level vulnerability that overcomes memory isolation allowing any user process to read the machine’s entire kernel memory
  - Hardware fixes in later processors, patches in some earlier ones (though concerns about performance penalties introduced), patched in OSes

Trust Examples

- Meltdown - hardware-level vulnerability that overcomes memory isolation allowing any user process to read the machine's entire kernel memory
  - Hardware fixes in later processors, patches in some earlier ones (though concerns about performance penalties introduced), patched in OSes
- Minimum Support Periods - Duration of software and security support
  - No requirement for operating system makers to provide this!
    - What are some arguments in favor? against?

Trust Examples

- Meltdown - hardware-level vulnerability that overcomes memory isolation allowing any user process to read the machine's entire kernel memory
  - Hardware fixes in later processors, patches in some earlier ones (though concerns about performance penalties introduced), patched in OSes
- Minimum Support Periods - Duration of software and security support
  - No requirement for operating system makers to provide this!
    - What are some arguments in favor? against?
- Therac-25 - A lethal race condition in the software of a medical radiation device
  - Where was the agential gullibility here? How could the engineers substituted that need for trust?

Thank you for all your hard work.
Best of luck on the midterm!!!

What questions do you have?