CS111 Final Review Session

Fall 2023

Wilmer Zuna, Isaac Gorelik

Key Topics

- Filesystems and Crash Recovery
- Multiprocessing and Pipes
- **Trust/ethics in systems**
- **Multithreading and Synchronization**
- **Dispatching and Scheduling**
- **Virtual Memory and Paging**

Multithreading and Synchronization

The Monitor Pattern: `ThreadPipe`

`ThreadPipe`

- Let's implement a class called `ThreadPipe`
- Like a pipe, but between threads instead of processes
- Main functionality:
  - `void put (char c);`
    - Puts 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

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

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

```cpp
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;
}
```

Key Question 1: Are there any race conditions possible? If so, how can we fix them?

`ThreadPipe`: Locked Implementation

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

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

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

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

`ThreadPipe`: Locked Implementation

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

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

Key Question 2: What if `ThreadPipe` is full/empty?

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

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

`ThreadPipe`: Locked Implementation

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

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

Key Question 2: What if `ThreadPipe` is full/empty?

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

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

`ThreadPipe`: Busy Waiting

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

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

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

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

`ThreadPipe`: Busy Waiting

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

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

Key Question 3: How can we avoid busy waiting?

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

char ThreadPipe::get() {
  lock.lock();
  while (count == 0) { // new
    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

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

  std::mutex lock;
  std::condition_variable added; // new
  std::condition_variable removed; // new
  char buffer[SIZE];
  int count = 0;
  int nextPut = 0;
  int nextGet = 0;
}
```

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

char ThreadPipe::get() {
  lock.lock();
  while (count == 0) {
    added.wait(lock); // new
  }
  count--;
  char c = buffer[nextGet];
  nextGet++;
  if (nextGet == SIZE) {
    nextGet = 0;
  }
  removed.notify_one(); // new
  lock.unlock();
  return c;
}
```

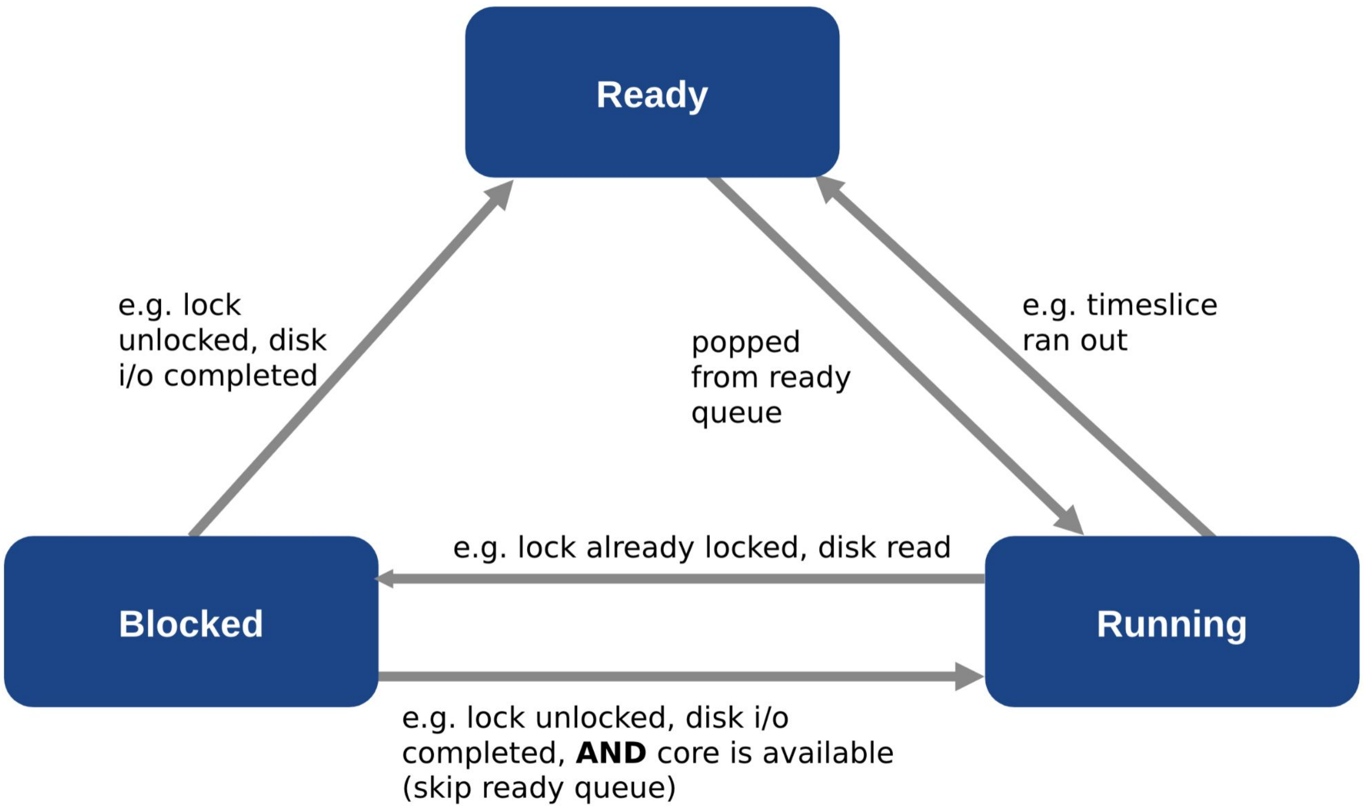
Dispatching and Scheduling

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.

Thread States



Virtual Memory

Different Approaches: Pros and Cons

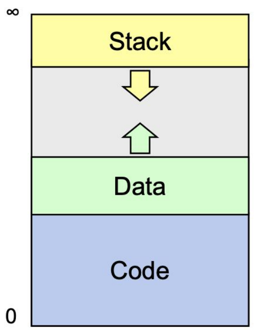
Approach 1: Base and Bound

Pros

- Quick address translation
- Doesn't require much space
- Separate virtual and physical address - can move physical memory, update base, etc.

Cons

- All memory allocated to a process must be contiguous virtual addresses
  - Stack often far from heap in virtual address space
- Can only grow upwards



Approach 2: Multiple Segments

Pros

- Quick address translation
- Little space needed per process
- Can allocate different discontinuous regions of virtual memory with different protections
  - Code
  - Heap
  - Stack

Cons

- Segments are of different sizes - will trend towards external fragmentation
- Segment encoding is limited

Approach 3: Paging

Pros

- Fixed page size - no external fragmentation
- Dynamically resize memory allocated to a process
- Grows in any direction
- Can assign different permissions to different pages
  - Code
  - Heap
  - Stack

Cons

- Internal fragmentation within pages - page size is 4KB, but may not need all memory.
- Slower/Complicated address translation
  - Clock algorithm
  - Thrashing - time spent in OS reading/writing pages to/from disk