CS 111 Final Review Session

Autumn 2022

Parthiv Krishna

Key Topics

- Filesystems and Crash Recovery
- Multiprocessing and Pipes
- **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
- `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;
};
```

```
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;
}
```

What if the ThreadPipe is full/empty?

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 added;
    std::condition_variable 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;
    }
    added.notify_one();
    lock.unlock();
}
```

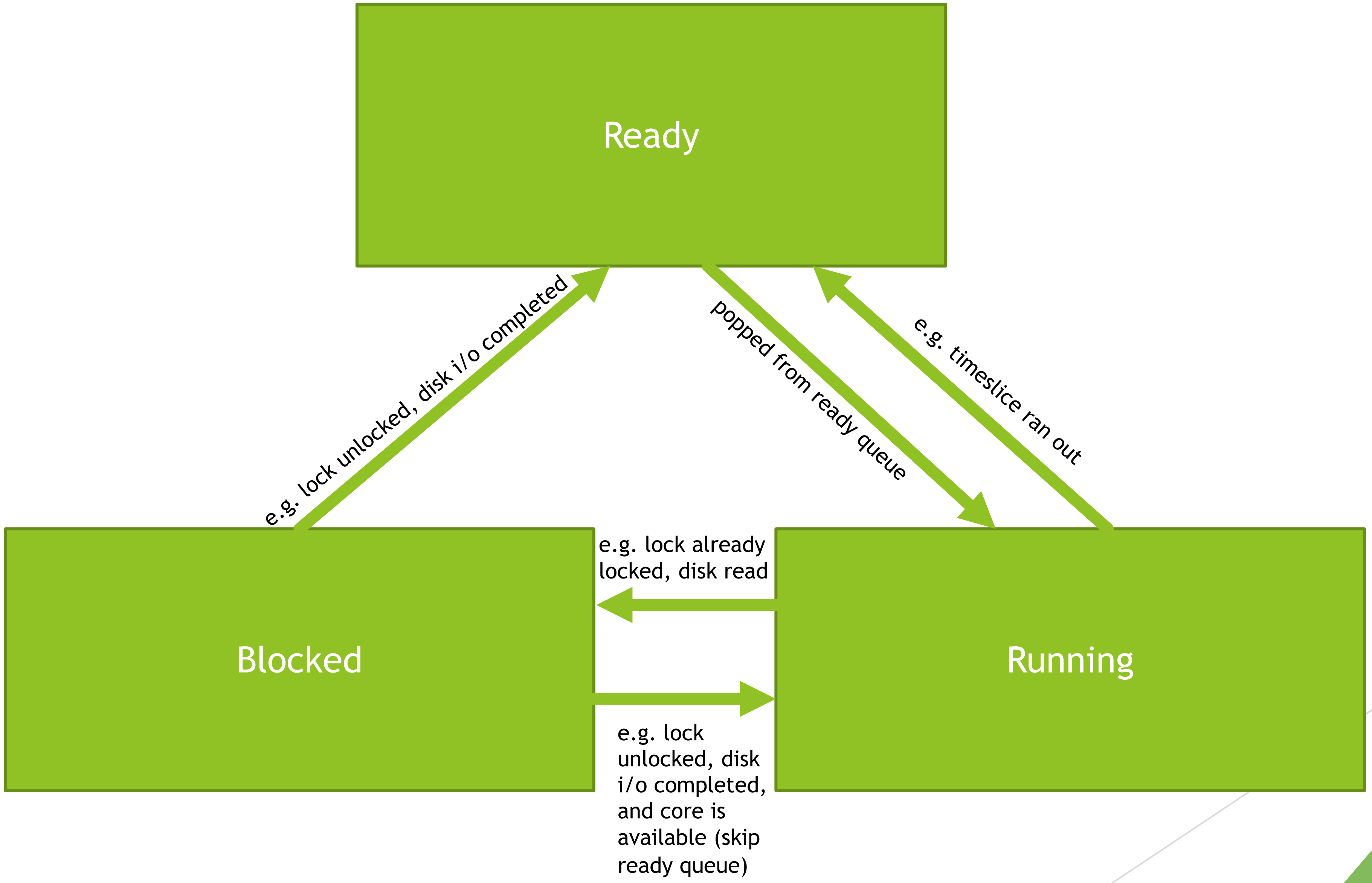
```
char Pipe::get() {
    lock.lock();
    while (count == 0) {
        added.wait(lock);
    }
    count--;
    char c = buffer[nextGet];
    nextGet++;
    if (nextGet == SIZE) {
        nextGet = 0;
    }
    removed.notify_one();
    lock.unlock();
    return c;
}
```

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?

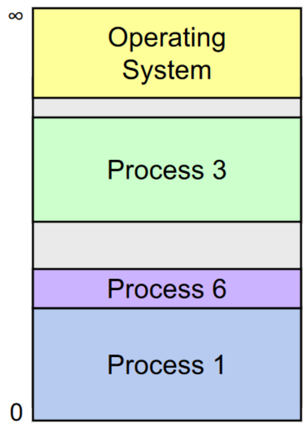
- Don’t want to run threads that can’t do any useful work right now (blocked).
- Ensures that we only run threads that can do something.


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)

Virtual Memory

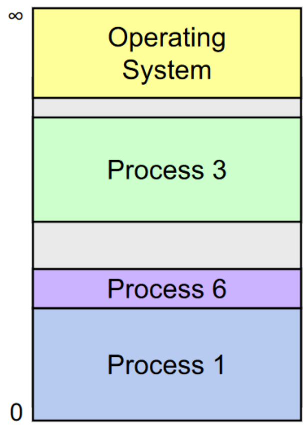
Different Approaches: Pros and Cons

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 information 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
    - Heap
    - 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
    - Heap
    - 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: memory management unit (MMU), translation lookaside buffer (TLB), page table walker (PTW)