Operating Systems

Futures

Futures

- Futures are a mechanism to synchronize on asynchronously produced data
- Recall that synchronization is closely related to scheduling
  - When a condition holds, a requesting thread can be scheduled; until then, it must sleep
  - The system orchestrates the correct semantics
- For the producer/consumer pattern, the consumer can read a future item and be awakened when one is produced

Futures

- The idea of a *promise* was developed by Friedman and Wise (Indiana University) in 1976
- The related idea of a *future* was proposed by Baker and Hewitt in 1977
- The terms are often used interchangeably or
  - a *promise* is the producer-side object – a container that is writable one time
  - a *future* is the consumer-side object – a placeholder for an object to be filled

http://en.wikipedia.org/wiki/Futures_and_promises

Empty/Full Memory

- An idea related to Futures is that of empty/full bits for memory locations
- The Tera (later Cray) MTA used memory like this
  - Threads were suspended when reading from an “empty” memory location
    - 128 hardware thread contexts – register sets – in the processor
- A Future blocks when a process reads while the memory location (variable) is empty
- Once filled (or if full already) the reader can consume the value and proceed

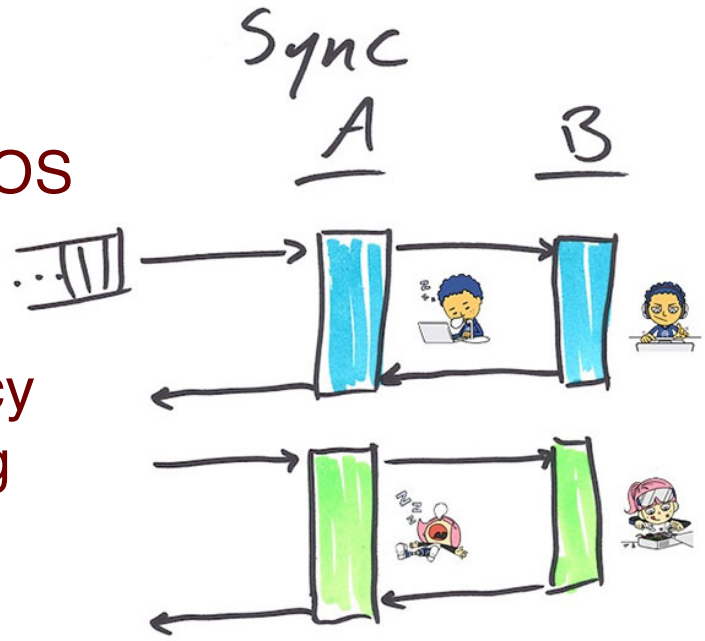
Facebook Futures

- Facebook has its own implementation of Futures and an interesting writeup that motivates their use
  - https://engineering.fb.com/2015/06/19/developer-tools/futures-for-c-11-at-facebook/
- Facebook's site, like most large Internet services, is comprised of smaller services — processing occurs like a "bucket brigade"
  - To expose parallelism — many concurrent activities
  - To improve scalability with asynchronous execution — minimize unnecessary synchronization

Why Asynchrony? (Facebook)

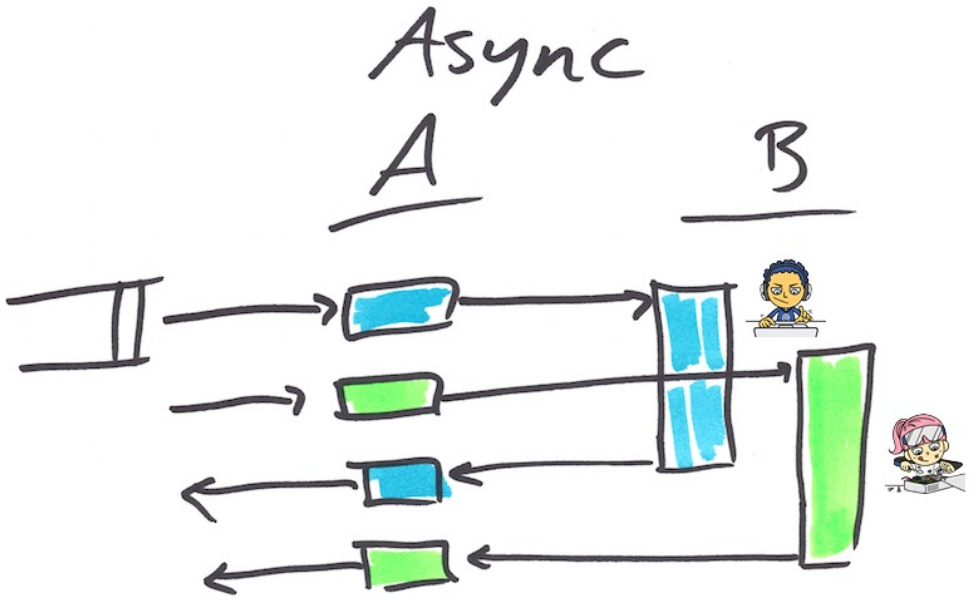
Consider a service *A* that talks to another service *B*. If *A* blocks while waiting for a reply from *B*, then *A* is *synchronous*. A blocked thread is idle; it cannot service other requests. Threads are heavyweight —

switching threads is inefficient, they have considerable memory overhead, and the OS will bog down if you make too many of them. The result is wasted resources, reduced throughput, and increased latency (because requests are in a queue, waiting to be serviced).



Why Asynchrony? (Facebook)

- It is more efficient to make service *A asynchronous*, meaning that while *B* is busy computing its answer, *A* has moved on to service other requests. When the answer from *B* becomes available, *A* will use it to finish the request.
- Traditional asynchronous code is more efficient than synchronous code, but it is not as easy to read
- Futures address this and Facebook's implementation adds composition (sequential and parallel)



Futures as a Language Construct

- C++11 provides a std::future class template
- An asynchronous operation can provide a future
  - A consumer can query, extract or wait for a future to be produced
- Also present in functional languages like Haskell
- Haskell defines an IVar, which is immutable and an MVar which is mutable
- We will call the immutable version “shared” later as it can be read multiple times

future.h

```
#ifndef _FUTURE_H_
#define _FUTURE_H_
typedef enum {
	FUTURE_EMPTY,
	FUTURE_WAITING,
	FUTURE_READY
} future_state_t;

typedef enum {
	FUTURE_EXCLUSIVE,
	FUTURE_SHARED,
	FUTURE_QUEUE
} future_mode_t;

typedef struct future_t {
	char *data;
	uint size;
	future_state_t state;
	future_mode_t mode;
	pid32 pid;
} future_t;
```

future.h (con't)

```
/* Interface for the Futures system calls */

future_t* future_alloc(future_mode_t mode, uint size, uint nelems);

syscall future_free(future_t*);

syscall future_get(future_t*, char*);

syscall future_set(future_t*, char*);


#endif /* _FUTURE_H_ */
```

Implementation Notes

- Examine the semaphore implementation
- Note: you may not directly use the semaphore implementation but it is not necessarily the most efficient way to do this
- Futures can drive the scheduler