Precept 3

COS 461

Concurrency is Useful

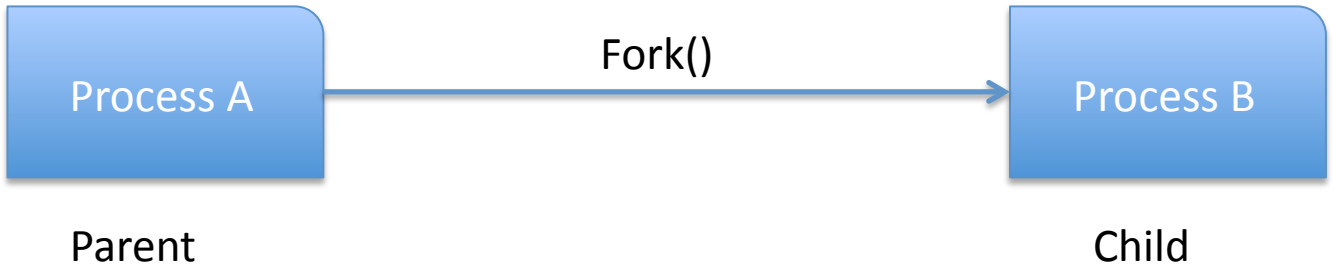
- Multi Processor/Core
- Multiple Inputs
- Don’t wait on slow devices

Methods for Concurrency

- **One process per client has disadvantages:**
  - High overhead – fork+exit ~ 100 µsec
  - Hard to share state across clients
  - Maximum number of processes limited
- **Concurrency through threads (MT)**
  - Data races and deadlock make programming tricky
  - Must allocate one stack per request
  - Many thread implementations block on some I/O or have heavy thread-switch overhead
- **Non-blocking read/write calls**
  - Unusual programming model

Fork() review

- It takes a process, clones its memory and starts a new process at a separate address space. (including file descriptors)
- IPC for parent <-> child communication
- Not very efficient (memory and time) for a lot of small requests - ~200 us



Threads

- Lightweight processes (10 – 100 x faster)
- Most memory is not copied
- Threads share: code, most data and file descriptors
- Unique thread ID, set of registers, stack pointer, local variable stack, return address, errno

Possible pitfalls of threads

- Racing conditions (on data)
  - Locking and synchronization required
- Thread-safe code (certain C library routines are not safe – e.g. strtok() vs strtok_r())
- Lookout for global or static variables

Pitfalls in Concurrency

- Deadlock: two processes block each other by holding onto resources that the other needs

- Livelock: processes change state but never progress (resource starvation)
  - Especially threads – why?

Efficient Concurency

- Have to control # of threads/processes:
  - thrashing: too many threads/processes contend and may place excessive load on (a) CPU (b) memory
  - too much time spent accessing the resource => minimal work done overall
- latency: creation/termination of threads/ processes can be expensive for each request

Thread/Proc. pool model

- At the beginning of the program we create a certain number of threads/proc.
- Keep track of threads that are busy / free by placing them in a ‘free’ queue
- Assign new requests to free threads
- Tuning can be used to optimize # of concurrent executions
  - prevent thrashing

Network Events – New Model

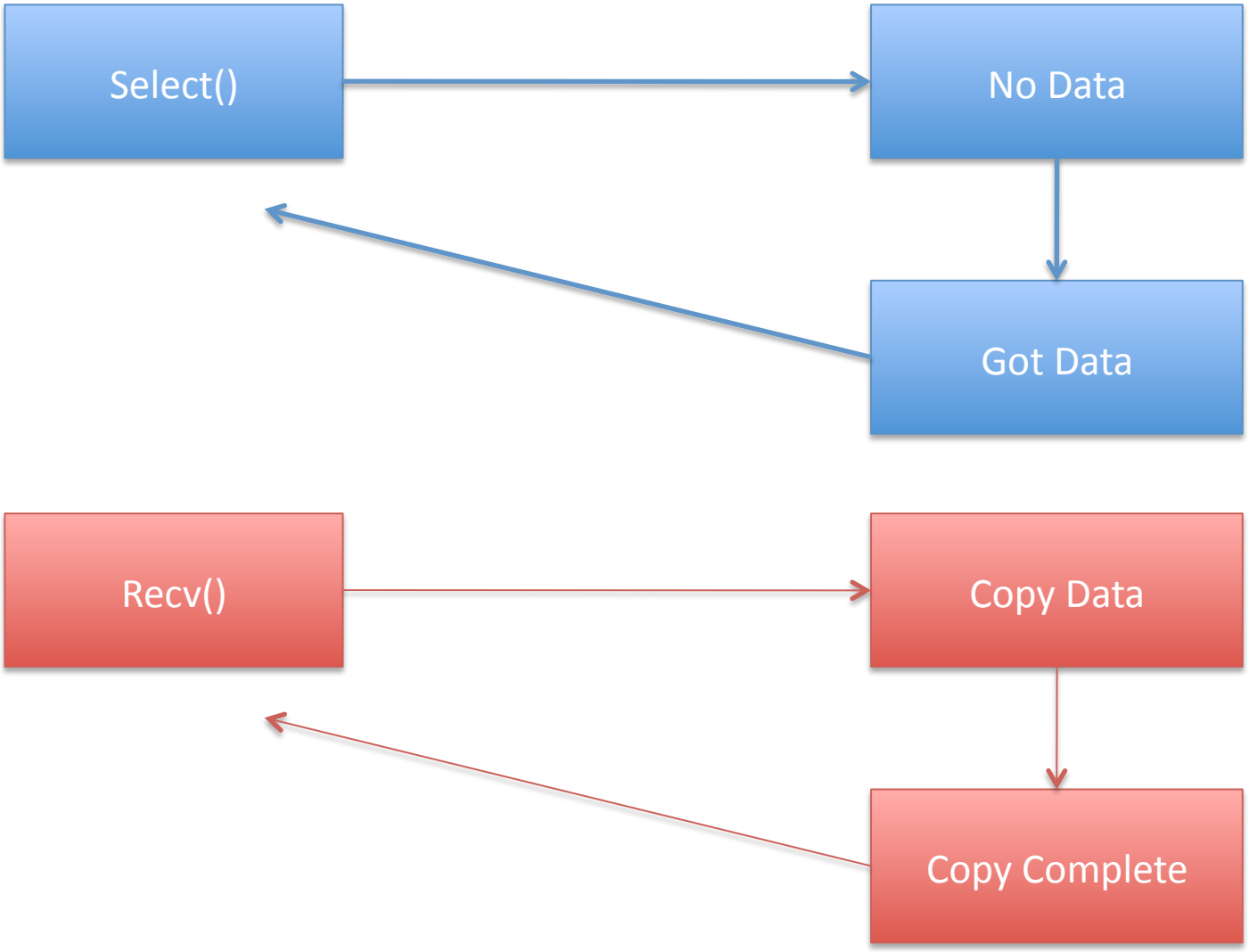
Old Model: multiple concurrent executions each of which use blocking calls

New Model: a single execution and a single call blocked over multiple sockets. Waits for any one of the sockets to have data.

non-blocking + select()

- non-blocking I/O
  - Do not wait for data/ability to write
    - If no data yet => return error.
  - e.g., recv() returns EWOULDBLOCK or EAGAIN if nothing to read
- select()
  - select() takes multiple descriptors and waits for one to be ready.
    - Note ability to wait on multiple descriptors at once
  - once ready, data operations return immediately
    - e.g. recv() returns the data immediately

Non Blocking IO + Select()



Async IO

- Similar to non-blocking IO, except a signal is delivered *after* copy is complete.
- Signal handler processes data.

Event Driven Programming

- Think about how your code looks with select()
  - Event selection: main loop
  - Event handling: call the matching event handler
- Flow of program is determined by events
  - Incoming packets, user clicks, hard-drive reads/writes
- "call-back”
  - Function pointer/code passed as argument to another function to be invoked later
  - What is your state during callback? Stack does not keep context state for you.

Backlog as flow control

- Backlog is the amount of outstanding connections that the server can queue in the kernel
- Limiting this control the rate at which servers can accept connections because any connection > backlog gets dropped
- Keep this in mind when we will see other types of flow control

Short Reads

- Why don't we read/write one char at a time?
- Why do we do buffering on read and write in the proxy?

A: each packet sent has headers which are overhead. High percentage in short packets