thr2csp

Toward Transforming Threads into Communicating Sequential Processes

Robert Lange and Spiros Mancoridis

Drexel University

Vision

Concurrent Programming

- Hardware speedup is slowing down, but platforms are becoming more concurrent.
- The degree of assurance and comprehension available for sequential programs today must be available for concurrent programs tomorrow.

GOAL: Improved program understanding, maintenance, and verification of concurrent programs

Problems with Shared-Memory Multithreading

Lee, E. 2006. The problem with threads. *Computer* 39, 5, 33-42.

- Built on a non-deterministic foundation
  - Multithreaded program execution is one (of many) interleavings of the statements of all threads
  - Determinism must be bolted on by the programmer
- Not composable
  - Entire program must be analyzed any time a thread is added or altered

Introduction to CSP with C++CSP

N. Brown and P. Welch, “An Introduction to the Kent C++ CSP Library,” Communicating Process Architectures, vol. 61, pp. 139-156, 2003.

CSP Solutions to the Problems with Threads

- CSP are deterministic by default
  - Non-determinism must be bolted on via choice constructs such as ALTing
  - Parallelism follows naturally from the network graph
- CSP are composable
  - Adding or altering one process cannot alter the behavior of another process
  - Each process can be analyzed independently

CSP

$$MyProcess = x1?t \rightarrow x2!(10 * (1 + t)) \rightarrow SKIP$$

A C++CSP Process

- Channels
- Run method
  - Fully Sequential
- Inputs
- Outputs

```
class MyProcess : public csp::CSProcess
{
private:
  csp::Chanin<int> x_in;
  csp::Chanout<int> x_out;
protected:
  void run()
  {
      int t;
      int __tmp_x;
      x_in.read(&t);
      t = 1 + t;
      t = 10 * t;
      x_out.write(&t);
  }
public:
  ...
};
```

A C++CSP Process

- Channels
- Run method
  - Fully Sequential
- Inputs
- Outputs

```
class MyProcess : public csp::CSProcess
{
private:
  csp::Chanin<int> x_in;
  csp::Chanout<int> x_out;
protected:
  void run()
  {
      int t;
      int __tmp_x;
      x_in.read(&t);
      t = 1 + t;
      t = 10 * t;
      x_out.write(&t);
  }
public:
  ...
};
```

A C++CSP Process

- Channels
- Run method
  - Fully Sequential
- Inputs
- Outputs

```
class MyProcess : public csp::CSProcess
{
private:
  csp::Chanin<int> x_in;
  csp::Chanout<int> x_out;
protected:
  void run()
  {
      int t;
      int __tmp_x;
      x_in.read(&t);
      t = 1 + t;
      t = 10 * t;
      x_out.write(&t);
  }
public:
  ...
};
```

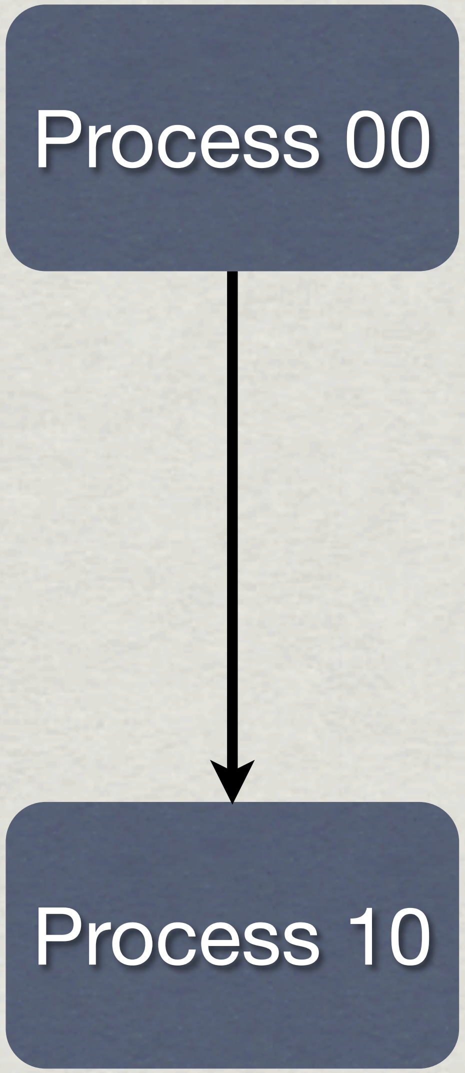
A C++CSP Process

- Channels
- Run method
  - Fully Sequential
- Inputs
- Outputs

```
class MyProcess : public csp::CSProcess
{
private:
   csp::Chanin<int> x_in;
   csp::Chanout<int> x_out;
protected:
   void run()
   {
       int t;
       int __tmp_x;
       x_in.read(&t);
       t = 1 + t;
       t = 10 * t;
       x_out.write(&t);
   }
public:
   ...
};
```

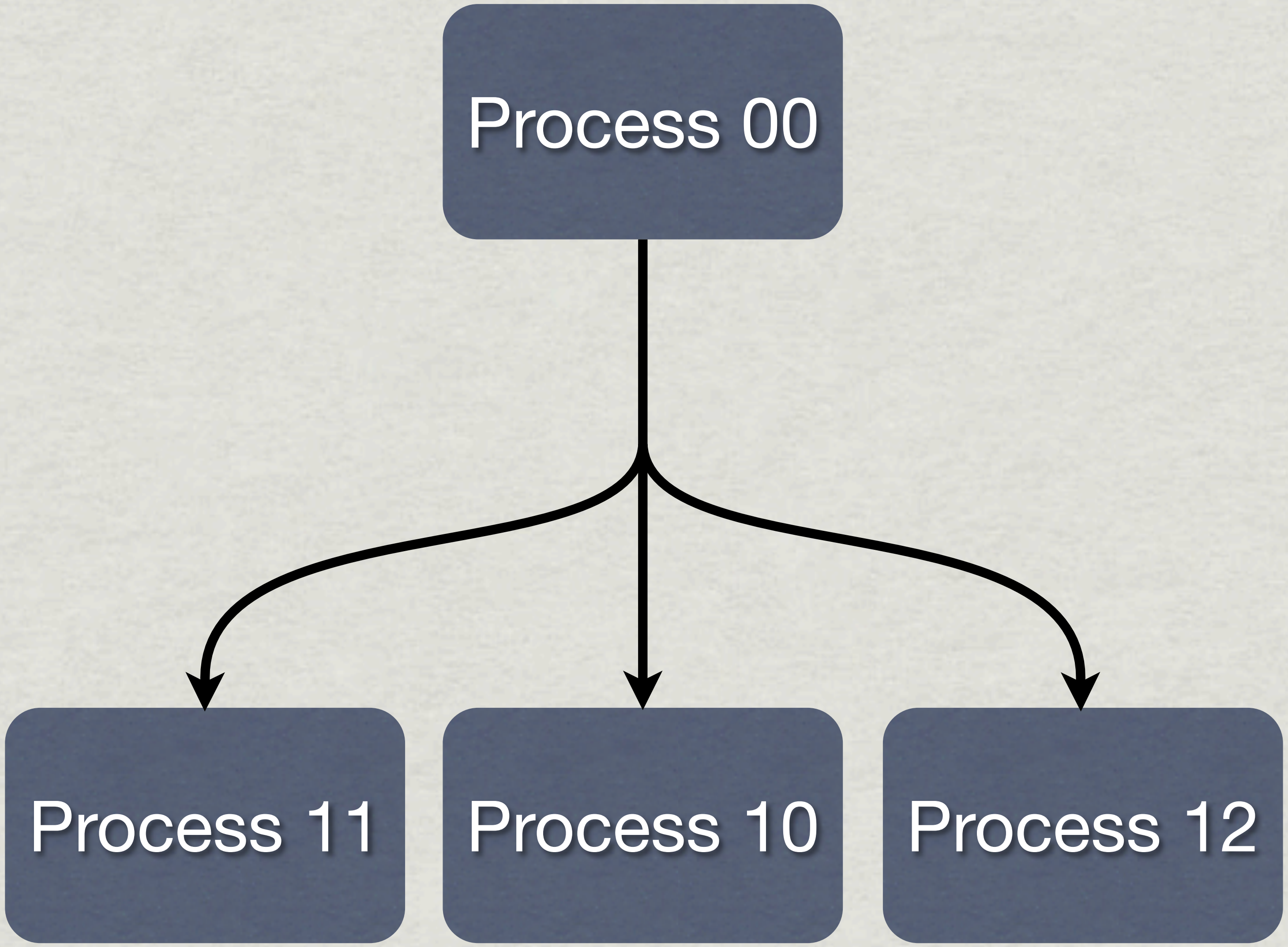
Communication Channels

- One2One
- One2Any
- Any2One
- Any2Any



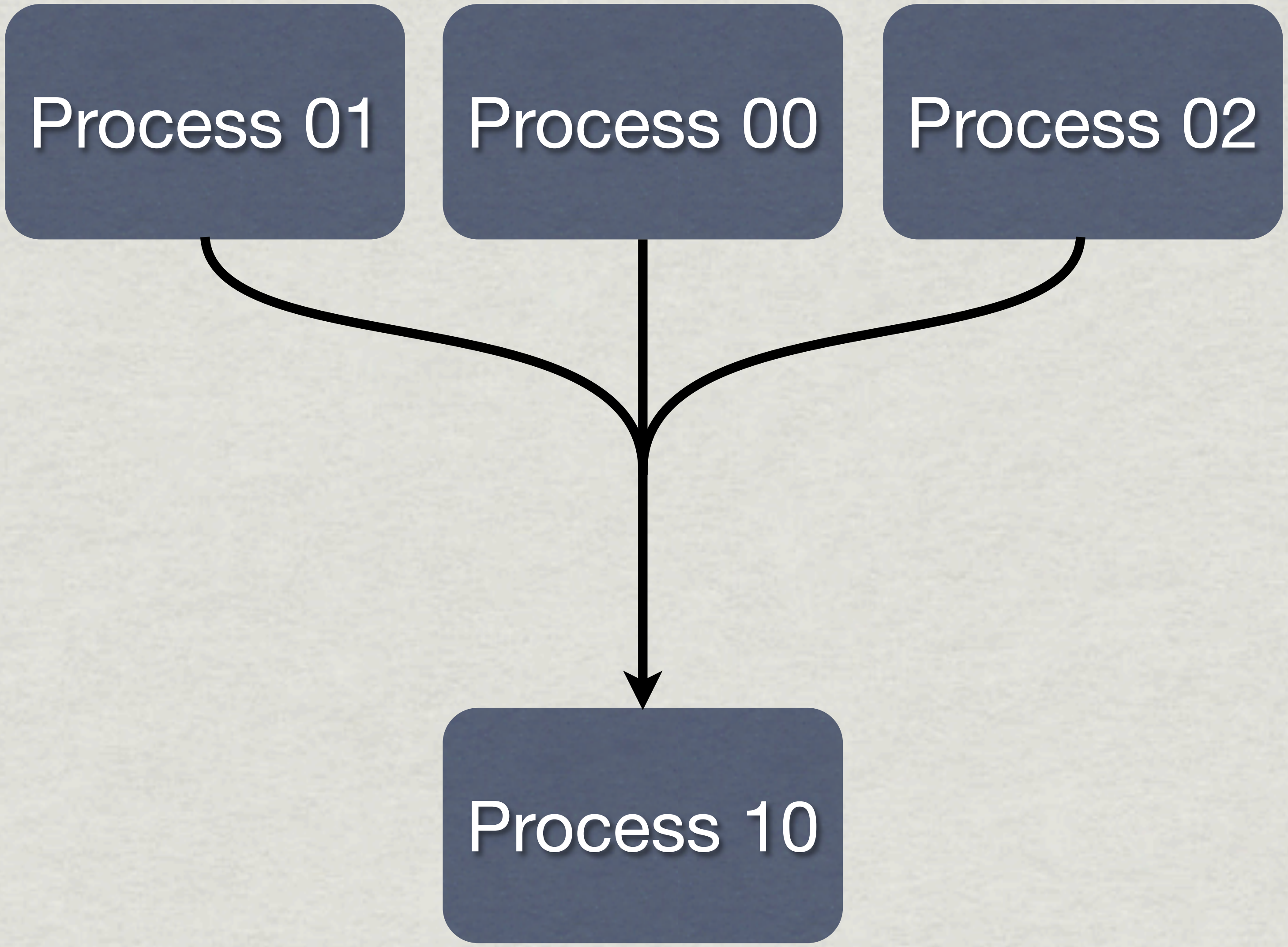
Communication Channels

- One2One
- ➡ One2Any
- Any2One
- Any2Any



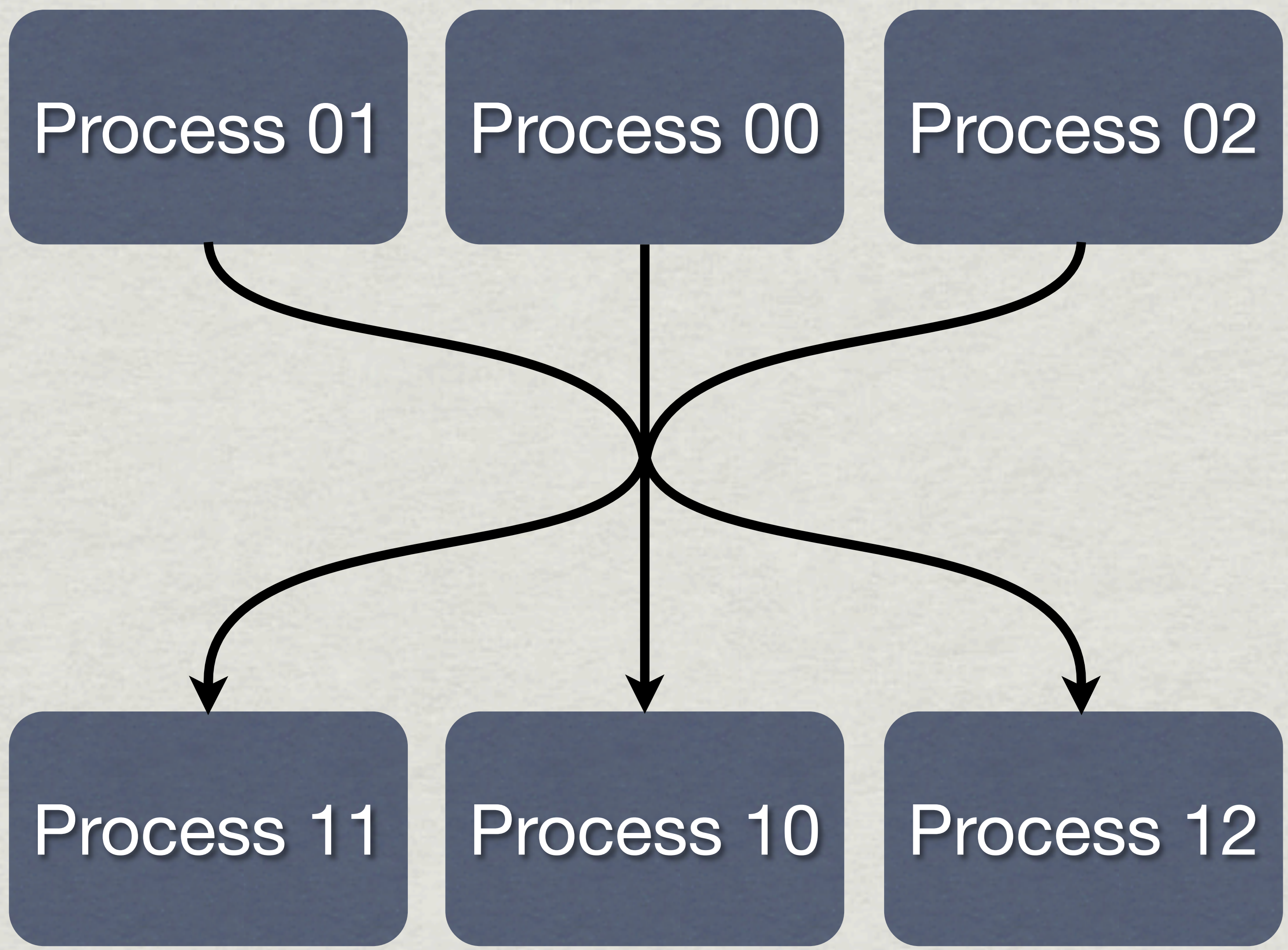
Communication Channels

- One2One
- One2Any
- Any2One
- Any2Any



Communication Channels

- One2One
- One2Any
- Any2One
- Any2Any



Communication Channels

- One2One
- One2Any
- Any2One
- Any2Any

- Synchronous
- Asynchronous
  - FIFO Blocking
  - Overwriting

Choice Among Channels

- Chooses which among the ready channels to select
- Selection strategies
  - Random
  - Round robin
  - Priority

```
list<Guard*> guards;

guards.push_back(chan1.inputGuard());
guards.push_back(chan2.inputGuard());

Alternative alt(guards);

int d;

while (true) {
  switch (alt.priSelect()) {
  case 0: // chan1
    chan1.read(&d);
    break;
  case 1: // chan2
    chan2.read(&d);
    break;
  }
}
```

Choice Among Channels

- Chooses which among the ready channels to select
- Selection strategies
  - Random
  - Round robin
  - Priority

```
list<Guard*> guards;

guards.push_back(chan1.inputGuard());
guards.push_back(chan2.inputGuard());

Alternative alt(guards);

int d;

while (true) {
  switch (alt.priSelect()) {
  case 0: // chan1
    chan1.read(&d);
    break;
  case 1: // chan2
    chan2.read(&d);
    break;
  }
}
```

Forking

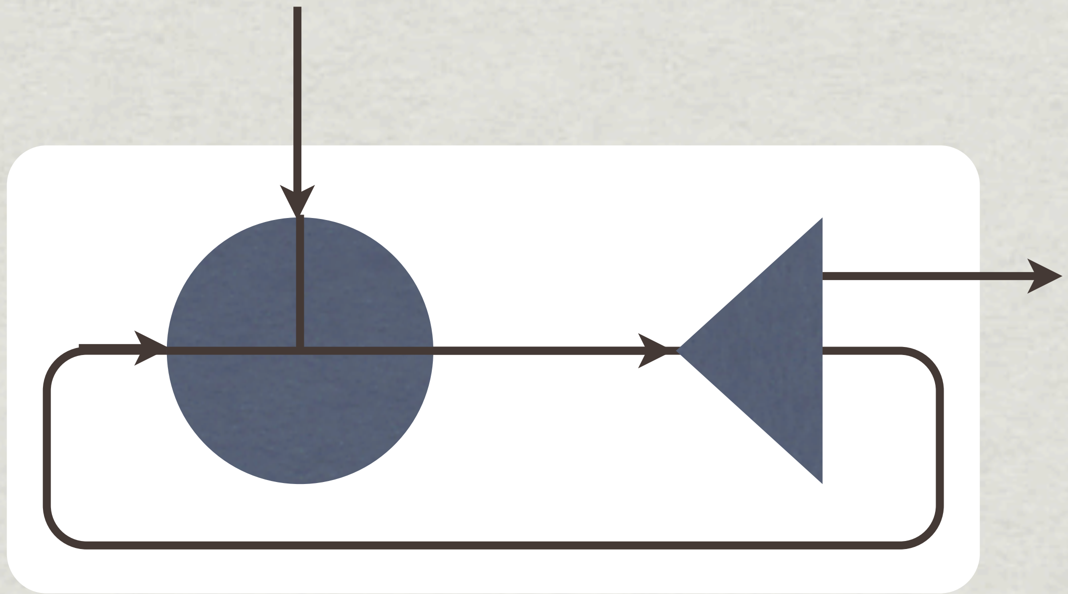
- ScopedForking enables asynchronous execution
- Calling process waits for the child process's termination when ScopedForking falls out of scope

```
ScopedForking* fork = new ScopedForking();

One2OneChannel<int> x;
MyProcess* myproc1;
MyProcess* myproc2;

myproc1 = new MyProcess(x.writer());
fork->fork(myproc1);

myproc2 = new MyProcess(x.reader());
fork->fork(myproc2);

delete fork;
```

Introducing the thread channels

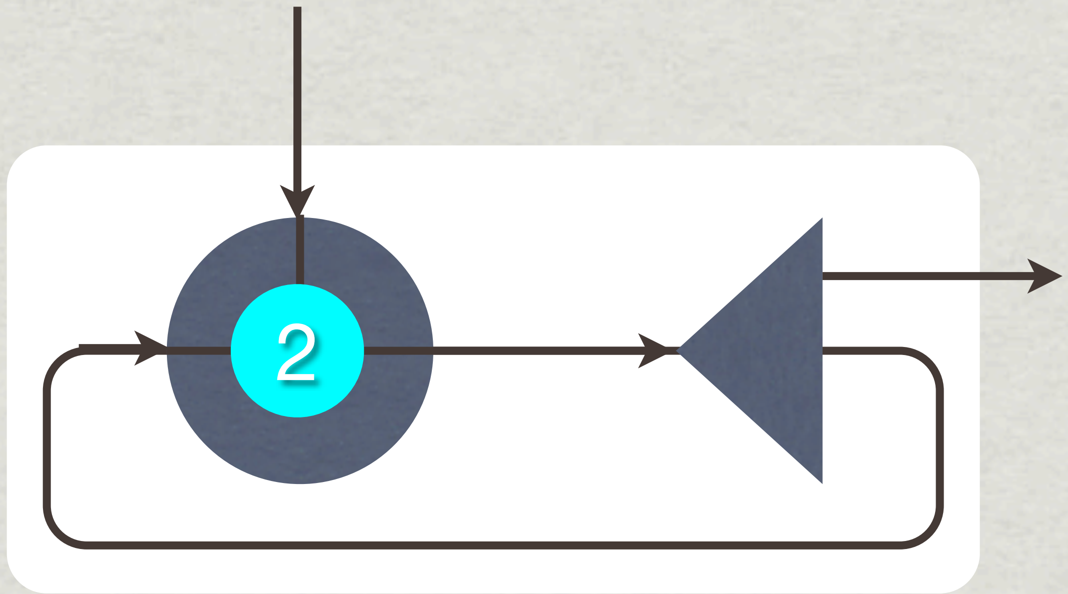
Shared Memory Channel (SHMChannel)

- Asynchronous Any2Any
- Buffer size = 1
- Overwriting
- Persistence -- reads do not remove data from channel



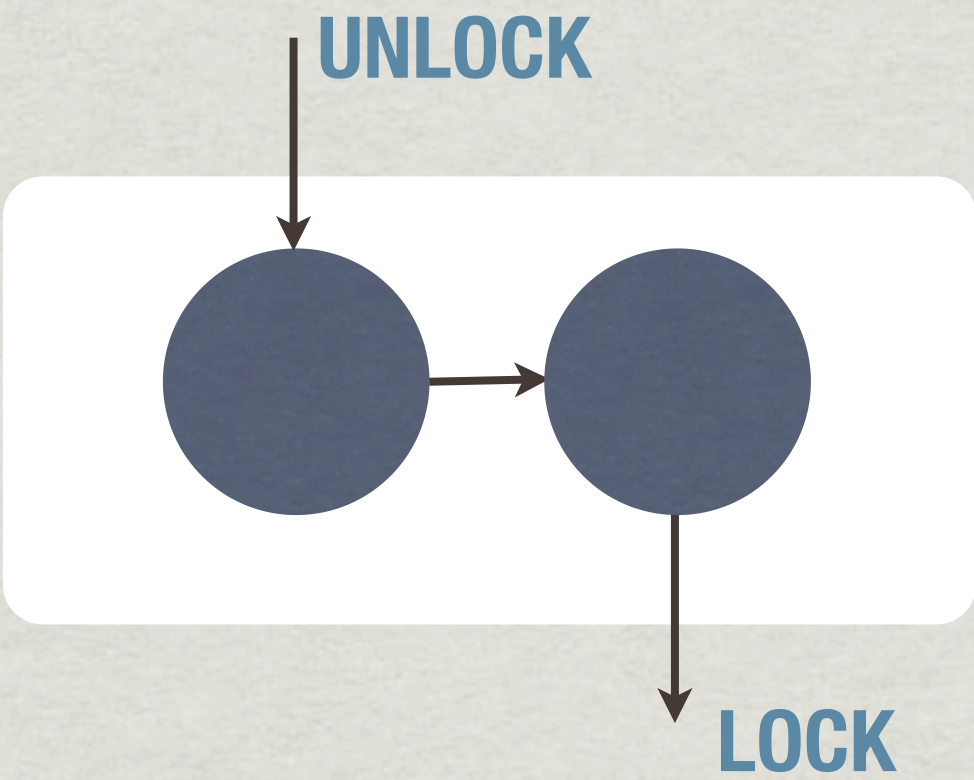
Shared Memory Channel (SHMChannel)

- Asynchronous Any2Any
- Buffer size = 1
- Overwriting
- Persistence -- reads do not remove data from channel



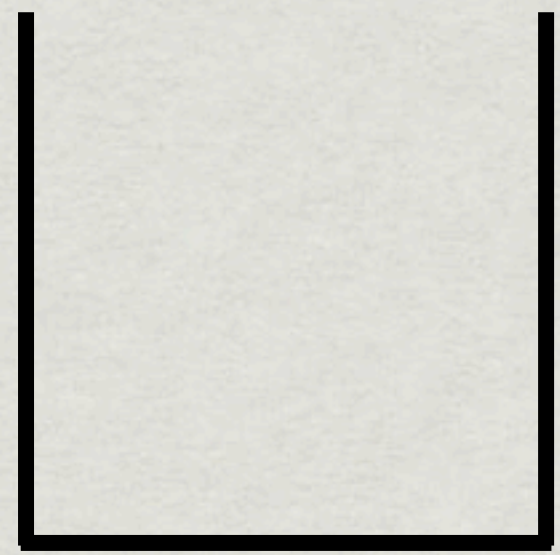
Lock Channel

* Lock
  * Reads token from channel
  * Blocks if no token
* Unlock
  * Writes token to channel
  * No effect if incorrect token is written



Signal Channel

- Bucket synchronization
- Wait
  - Fall into bucket
- Signal
  - Empty bucket

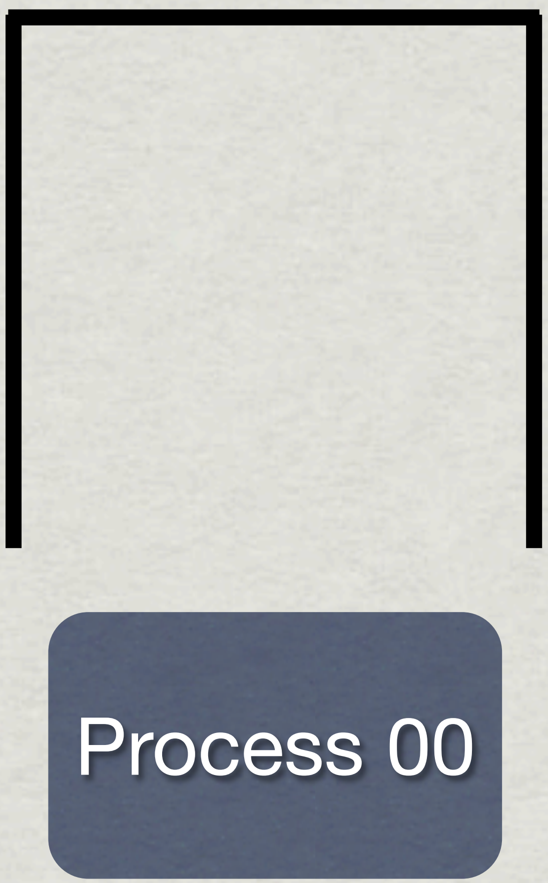


Signal Channel

- Bucket synchronization
- Wait
  - Fall into bucket
- Signal
  - Empty bucket

Signal Channel

- Bucket synchronization
- Wait
  - Fall into bucket
- Signal
  - Empty bucket



Strategies to implement threading as CSP

Steps

1. Create processes
   * Identify thread entry functions
2. Create channels to link processes together
   * Identify shared variables (or structs)
3. Transform shared variable accesses into reads/writes on SHMChannels
4. Handle synchronization
   * mutexes and condition variables

1. Which functions are threads?

- In POSIX threading, there is no keyword to denote a thread
- Any function with the correct prototype can be a thread start function
- Must locate thread start functions called via the `pthread_create` function

```
void* mythr(void*)
```

```
pthread_create(?_,?_,mythr,?_)
```

2. What variables does a thread access?

- Simplifying assumption: Assume no global shared variables
- Once again, `pthread_create` holds the answer

```
typedef struct {
  pthread_mutex_t xm;
  pthread_cond_t xcv;
  int xst;
  int x;
} shared_t;

int main(...) {
  shared_t s;

  pthread_create(?_,?_,?_,&s);
```

2. Transforming shared variables to thread channels

```
typedef struct {
  pthread_mutex_t xm;
  pthread_cond_t xcv;
  int xst;
  int x;
} shared_t;

int main(...) {
  shared_t s;

  pthread_create(?_,?_,?_,&s);
```

2. Transforming shared variables to thread channels

```
typedef struct {
  pthread_mutex_t xm;
  pthread_cond_t xcv;
  int xst;
  int x;
} shared_t;

int main(....) {
  shared_t s;

  pthread_create(?_,?_,?_,&s);
```

```
int main(...) {
  LockChannel s_xm;
  SignalChannel s_xcv;
  SHMChannel<int> s_xst;
  SHMChannel<int> s_x;

  pthread_create(?,?,?,&s);
```

3. Transforming the thread start function declaration

```
void* partA(void* arg)
{
   ...
}
```

3. Transforming the thread start function declaration

```
void* partA(void* arg)
{
   ...
}
```

```
class partA : public CSProcess
{
private:
  Chanin<int> xm_in;    Chanout<int> xm_out;
  Chanin<int> xcv_in;   Chanout<int> xcv_out;
  Chanin<int> xst_in;   Chanout<int> xst_out;
  Chanin<int> x_in;     Chanout<int> x_out;
protected:
  void run()
  {
    ...
  }
public:
  [redacted]
  {}
};
```

Censored: Ugly C++ boilerplate

3. Transforming the thread start function declaration

```
void* partA(void* arg)
{
  ...
}
```

```
class partA : public CSProcess
{
private:
  Chanin<int> xm_in;    Chanout<int> xm_out;
  Chanin<int> xcv_in;   Chanout<int> xcv_out;
  Chanin<int> xst_in;   Chanout<int> xst_out;
  Chanin<int> x_in;     Chanout<int> x_out;
protected:
  void run()
  {
    ...
  }
public:
  [redacted]
  {}
};
```

Censored: Ugly C++ boilerplate

3. Transforming the thread start function declaration

```
void* partA(void* arg)
{
   ...
}
```

```
class partA : public CSProcess
{
private:
  Chanin<int> xm_in;    Chanout<int> xm_out;
  Chanin<int> xcv_in;   Chanout<int> xcv_out;
  Chanin<int> xst_in;   Chanout<int> xst_out;
  Chanin<int> x_in;     Chanout<int> x_out;
protected:
  void run()
  {
    ...
  }
public:
  partA(const Chanin<int>& _xm_in, const Chanout<int>& _xm_out,
[redacted]
  {}
};
```

Censored: Ugly C++ boilerplate

3 & 4. Transforming thread bodies

```
void* partA(void* arg)
{
  int t;
  shared_t* s = (shared_t*) arg;

  pthread_mutex_lock(&s->xm);
  s->x = 1 + s->x;
  s->xst = 1;
  pthread_cond_signal(&s->xcv);
  pthread_mutex_unlock(&s->xm);
}
```

```
void run()
{
  int lcl_x;
  int lcl_xst;
  int lcl_xm
  int lcl_xcv;

  xm_in.read(&lcl_xm);
  x_in.read(&lcl_x);
  lcl_x = 1 + lcl_x;
  x_out.write(&lcl_x);
  lcl_xst = 1;
  xst_out.write(&lcl_xst);
  lcl_xcv = 1;
  xcv_out.write(&lcl_xcv);
  xm_out.write(&lcl_xm);
}
```

3 & 4. Transforming thread bodies

```
void* partA(void* arg)
{
  int t;
  shared_t* s = (shared_t*) arg;

  pthread_mutex_lock(&s->xm);
  s->x = 1 + s->x;
  s->xst = 1;
  pthread_cond_signal(&s->xcv);
  pthread_mutex_unlock(&s->xm);
}
```

```
void run()
{
  int lcl_x;
  int lcl_xst;
  int lcl_xm
  int lcl_xcv;

  xm_in.read(&lcl_xm);
  x_in.read(&lcl_x);
  lcl_x = 1 + lcl_x;
  x_out.write(&lcl_x);
  lcl_xst = 1;
  xst_out.write(&lcl_xst);
  lcl_xcv = 1;
  xcv_out.write(&lcl_xcv);
  xm_out.write(&lcl_xm);
}
```

3 & 4. Transforming thread bodies

```
void* partA(void* arg)
{
  int t;
  shared_t* s = (shared_t*) arg;

  pthread_mutex_lock(&s->xm);
  s->x = 1 + s->x;
  s->xst = 1;
  pthread_cond_signal(&s->xcv);
  pthread_mutex_unlock(&s->xm);
}
```

```
void run()
{
  int lcl_x;
  int lcl_xst;
  int lcl_xm
  int lcl_xcv;

  xm_in.read(&lcl_xm);
  x_in.read(&lcl_x);
  lcl_x = 1 + lcl_x;
  x_out.write(&lcl_x);
  lcl_xst = 1;
  xst_out.write(&lcl_xst);
  lcl_xcv = 1;
  xcv_out.write(&lcl_xcv);
  xm_out.write(&lcl_xm);
}
```

3 & 4. Transforming thread bodies

```
void* partA(void* arg)
{
  int t;
  shared_t* s = (shared_t*) arg;

  pthread_mutex_lock(&s->xm);
  s->x = 1 + s->x;
  s->xst = 1;
  pthread_cond_signal(&s->xcv);
  pthread_mutex_unlock(&s->xm);
}
```

```
void run()
{
  int lcl_x;
  int lcl_xst;
  int lcl_xm
  int lcl_xcv;

  xm_in.read(&lcl_xm);
  x_in.read(&lcl_x);
  lcl_x = 1 + lcl_x;
  x_out.write(&lcl_x);
  lcl_xst = 1;
  xst_out.write(&lcl_xst);
  lcl_xcv = 1;
  xcv_out.write(&lcl_xcv);
  xm_out.write(&lcl_xm);
}
```

Strategies to improve the quality of generated CSP

BETA!

Process Simplification Strategies

- Predicated wait simplification
- Unused read elimination
- Empty lock elimination

Process Simplification

```
xm_in.read(&lcl_lock);
xst_in.read(&lcl_xst);
if (1 != lcl_xst) {
  xm_out.write(&lcl_xm);
  xcv_in.read(&lcl_xcv);
  xm_in.read(&lcl_xm);
}
...
xm_out.write(&lcl_xm);
```

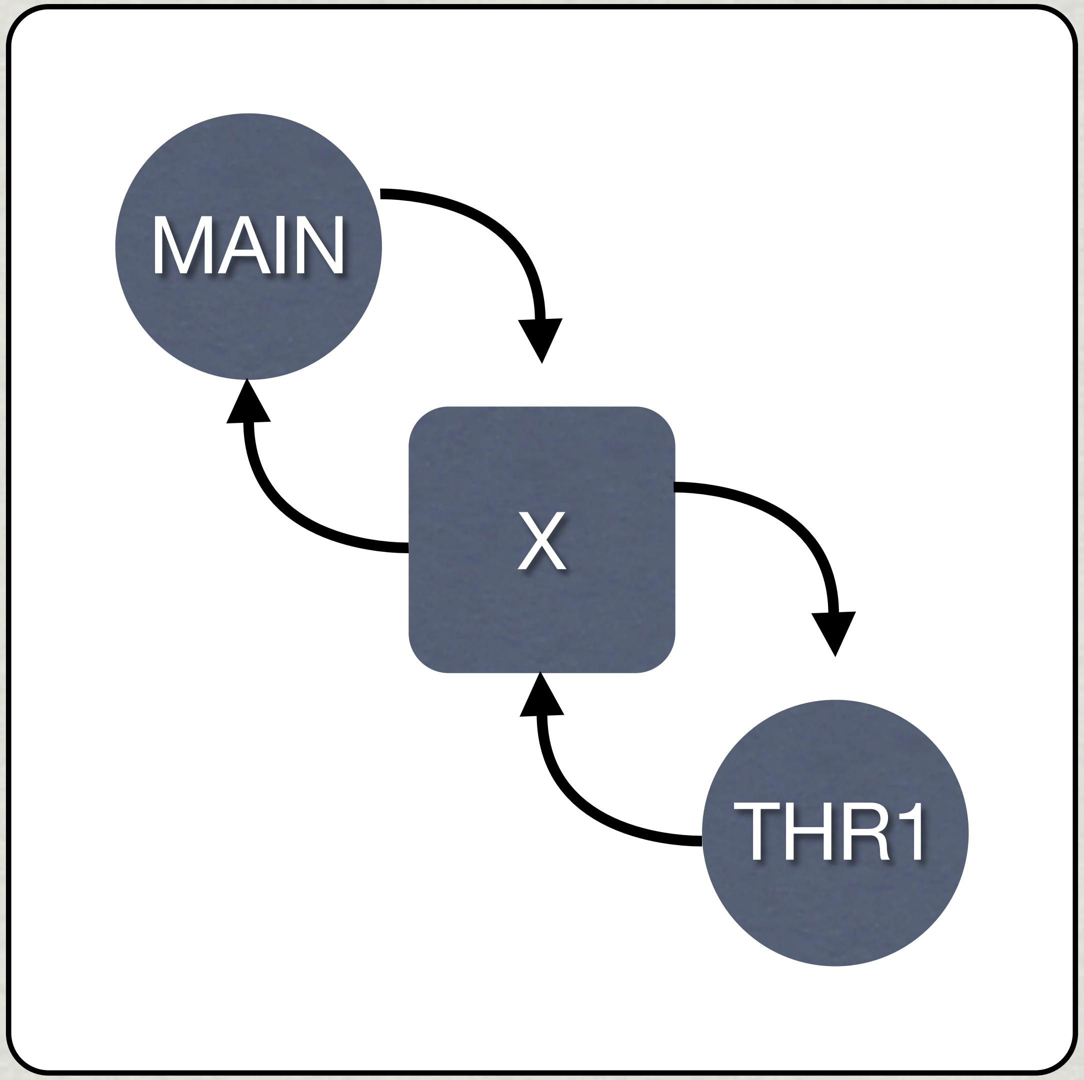
```
pthread_mutex_lock(s->xm);
if (1 != s->xst) {
  pthread_cond_wait(s->xcv, s->xm);
}
...
pthread_mutex_unlock(s->xm);
```

```
xcv_in.read(&lcl_xcv);
xm_in.read(&lcl_xm);
...
xm_out.write(&lcl_xm);
```

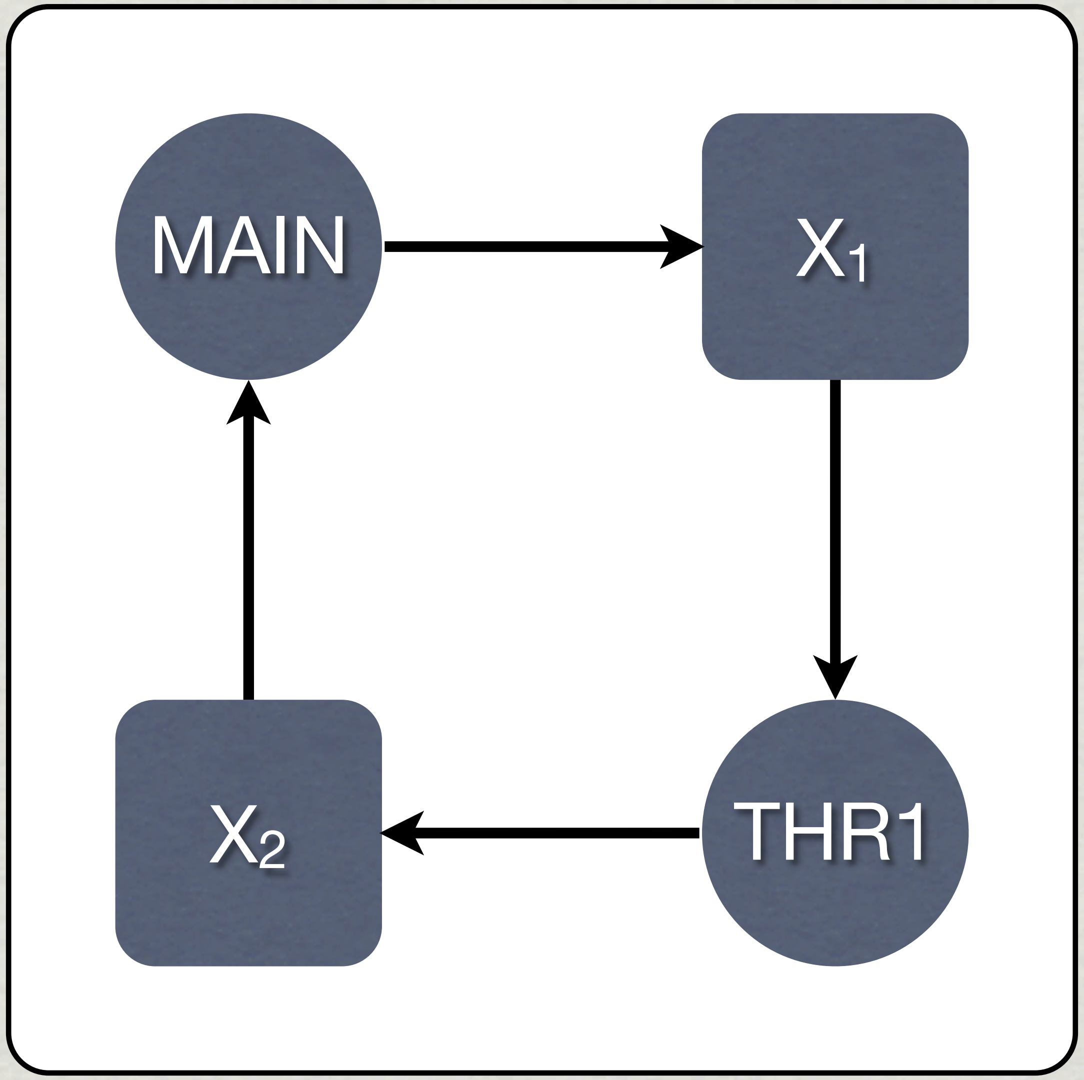
Process Network Simplification Strategies

- Multiple-read/write elimination / cycle elimination
  - Channel splitting
  - Process splitting
- Shared Memory Channel Conversion
- Lock and Signal Channel Elimination

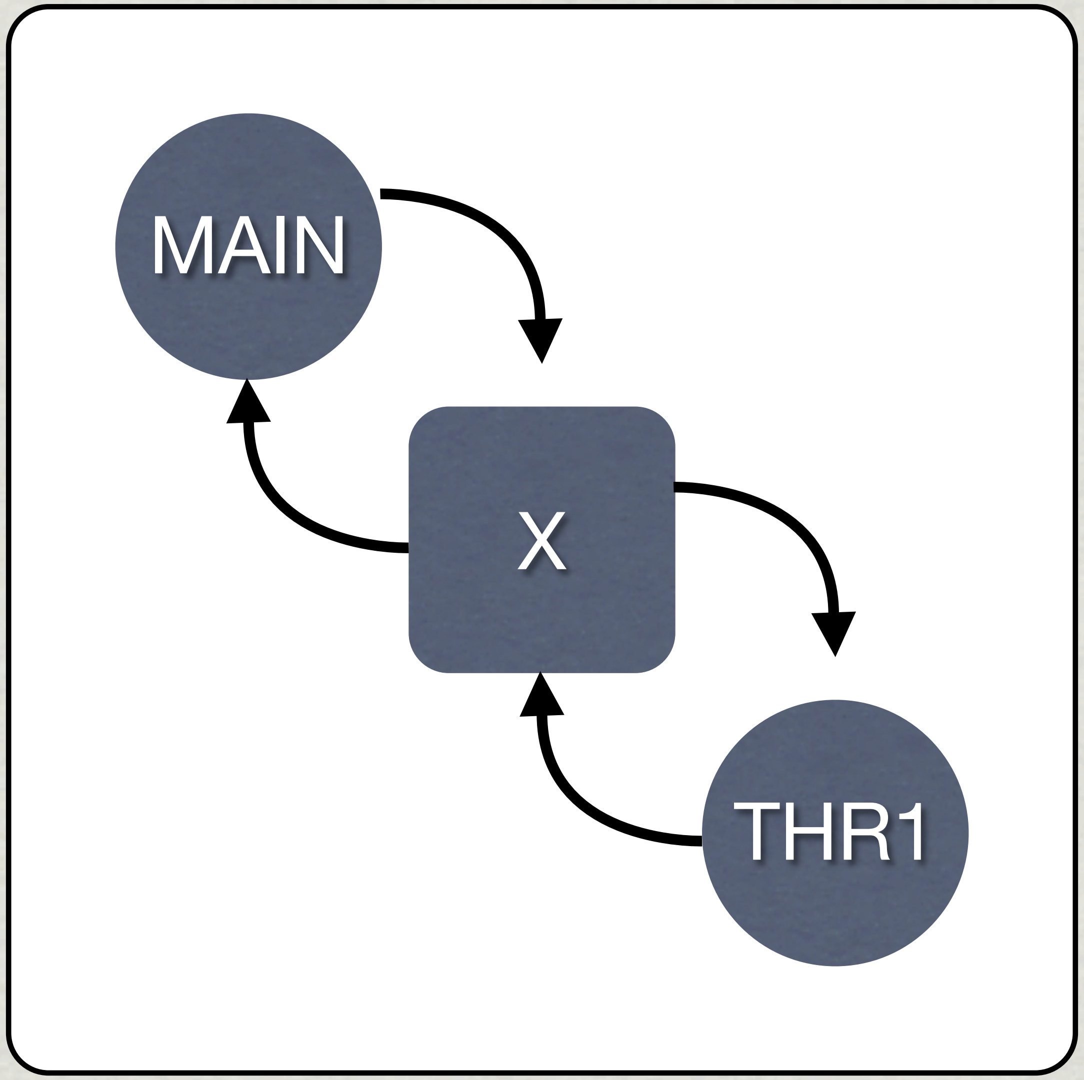
SHMChannel Splitting



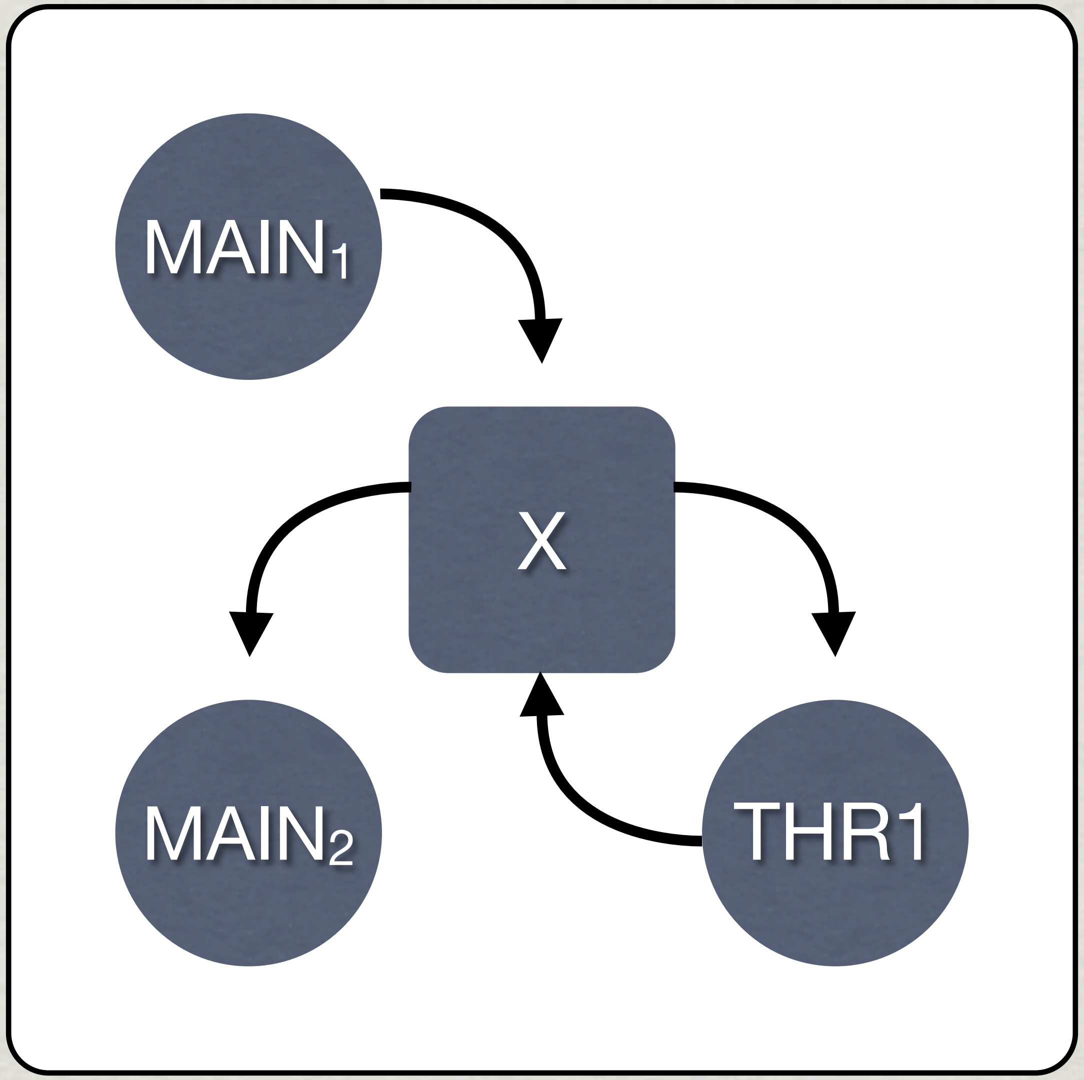
SHMChannel Splitting



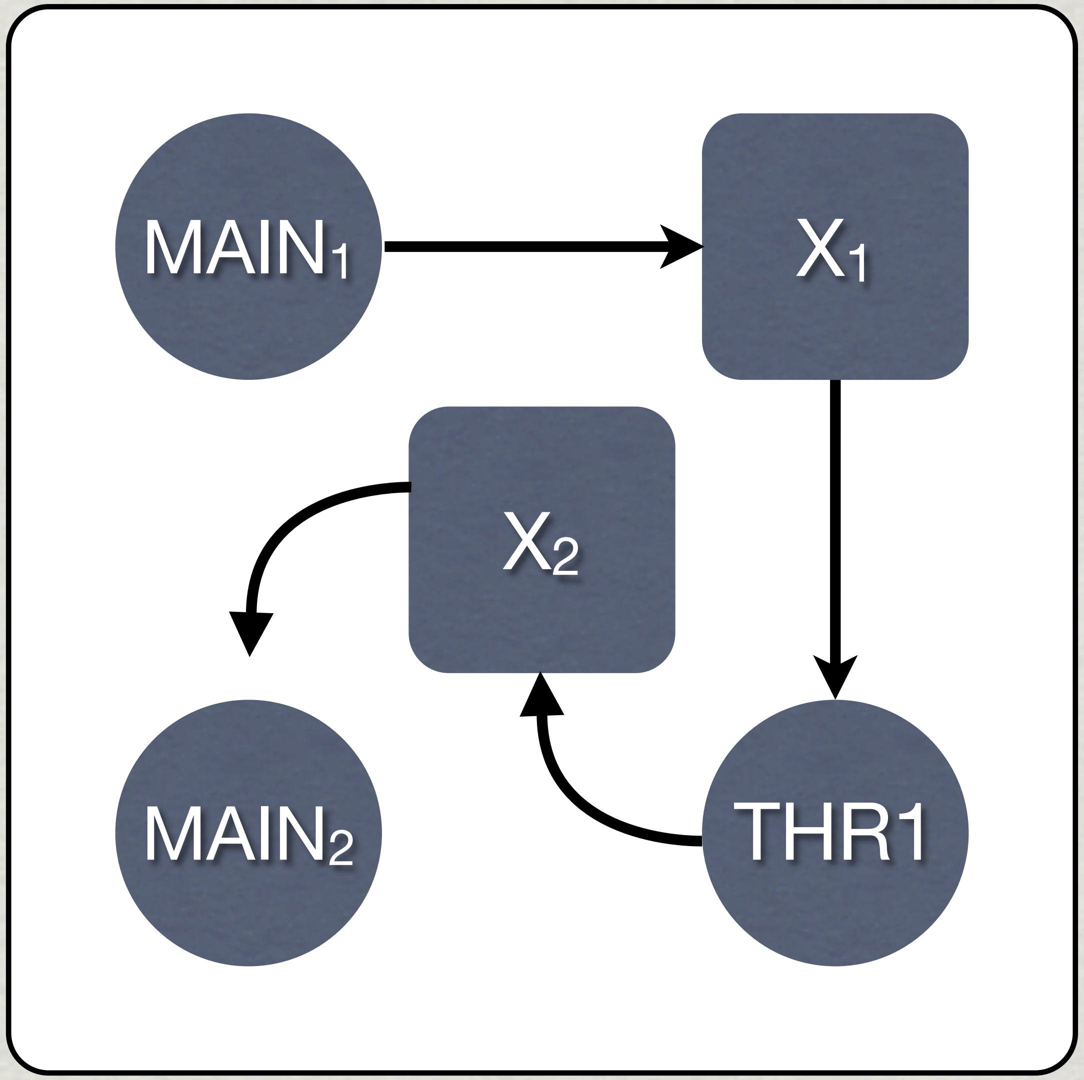
Process Splitting



Process Splitting



Channel + Process Splitting



Wrap-up

- Threads are problematic
- Threads can be implemented as CSP with functional equivalence
- Simplification steps can reduce complexity of resulting CSP code semi-automatically

Future Work

- Handle sources of side-effects more intelligently (e.g., shared pointers)
- Strategies to convert shared memory channels to proper CSP channels and eliminate explicit synchronization
  - Tools to provide user assistance

Controversy!

- Multithreading will be an obstacle to deploying safe, stable, highly concurrent programs.
  - Wait, that's probably not very controversial...
  - Multithreading is to concurrency what assembly language is to programming.
- Incurring the cost of converting legacy threaded programs into CSP-style programs may provide long-term benefits of improved maintainability and improved opportunity for parallelism.

Thank you.