Week 11: Memory and Thread programming

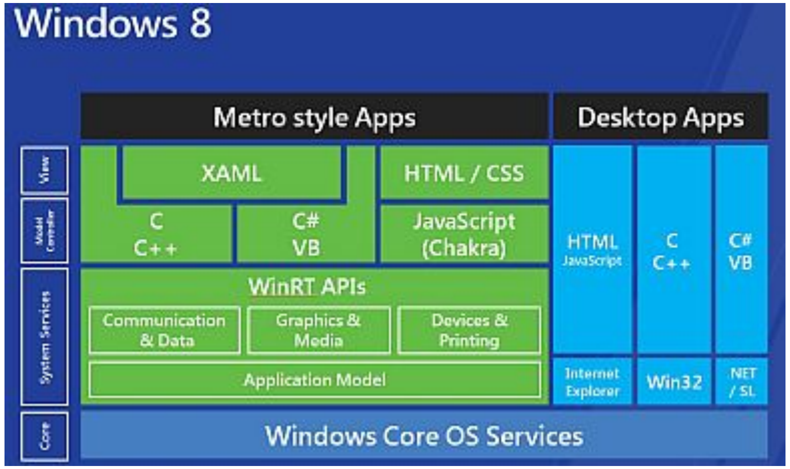
COMP2129

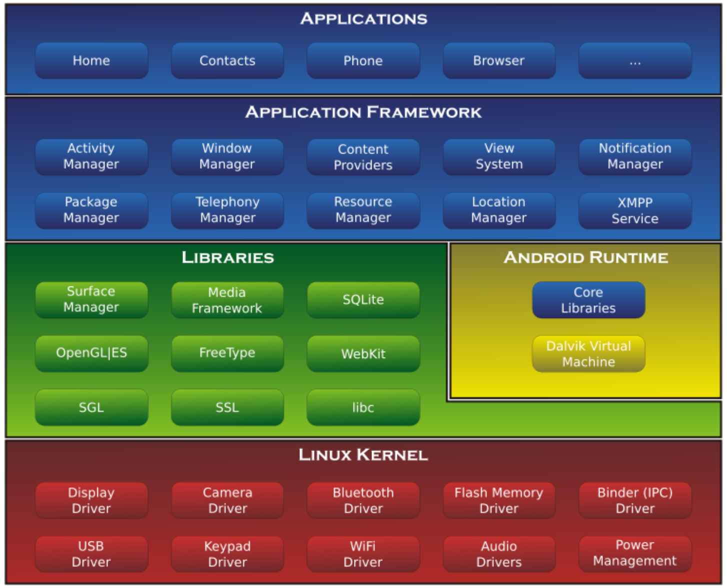
Outline

- Processes and memory
  - Hierarchy
  - Memory access patterns
- The thread pool

Processes and Memory

- Operating System (OS) only purpose is make the software run on the hardware
- OS is an overhead cost. Everything that the OS does requires resources from the hardware: memory, computation
- OS is a necessary abstraction for program writers so they don’t need to know hardware details





Processes and Memory

- OS manages all the memory for processes (execution state of a program), devices and communication (interrupts).

- OS will computationally solve many problems (search tasks) that programmer doesn’t have to worry about.

- To do this, OS also needs to have additional memory for each program and for itself.

- Memory contains both ***data*** and ***instructions*** (binary code).

Processes and Memory

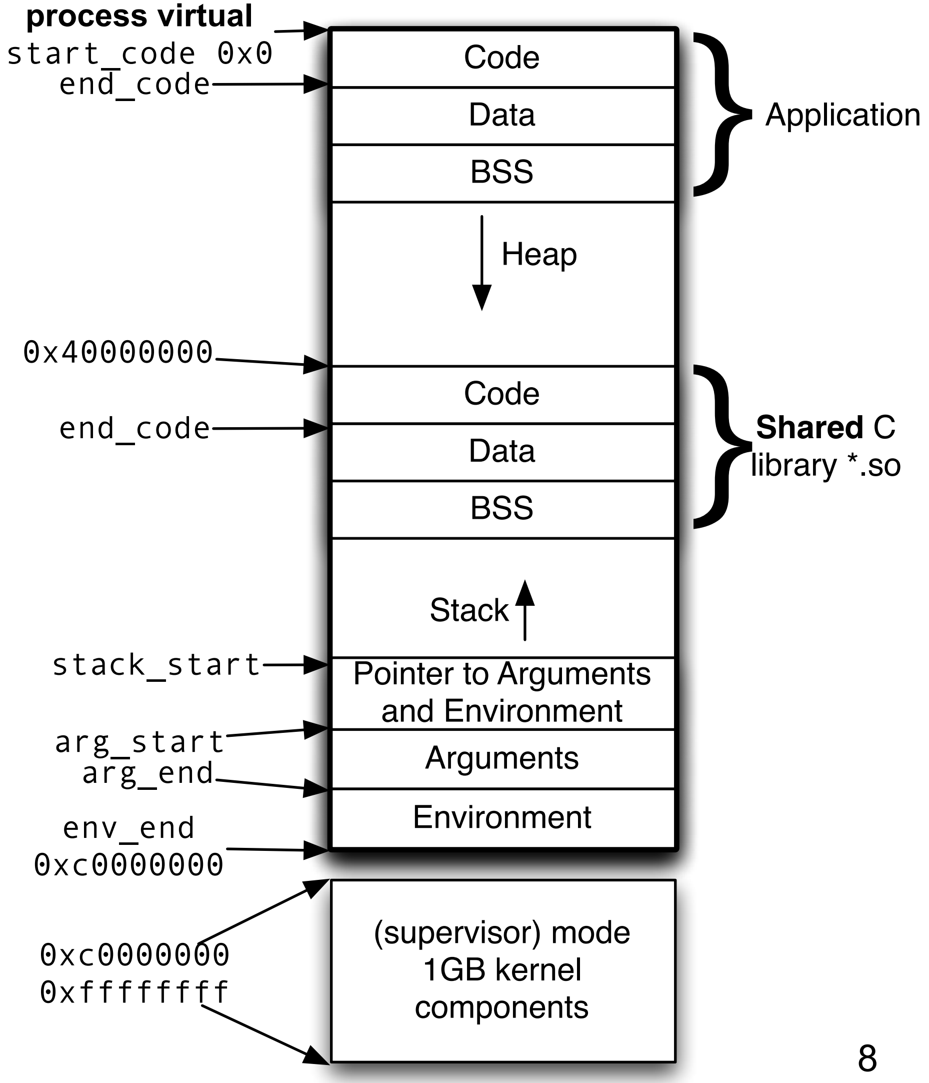
- System memory is divided into two spaces
- Kernel
  - only processes with certain privileges can r/w/x
  - OS functions and data live here e.g. I/O, processes, devices
  - protect the hardware by only accessing through this layer
- User
  - all user created processes privilege depends on who created
  - These programs are treated as rogue/untrusted that can run and die
  - processes in this space are independent
- User space processes use *system calls* to access Kernel space *

Processes and Memory

- User creating a new process
  - OS creates a new image of memory that will be used by the process by cloning an existing process.
  - This has permissions associated with r/w/x of user/group
- The memory inside the process is assigned a virtual address range
  - Pieces of virtual memory get mapped to physical memory when they are needed during execution

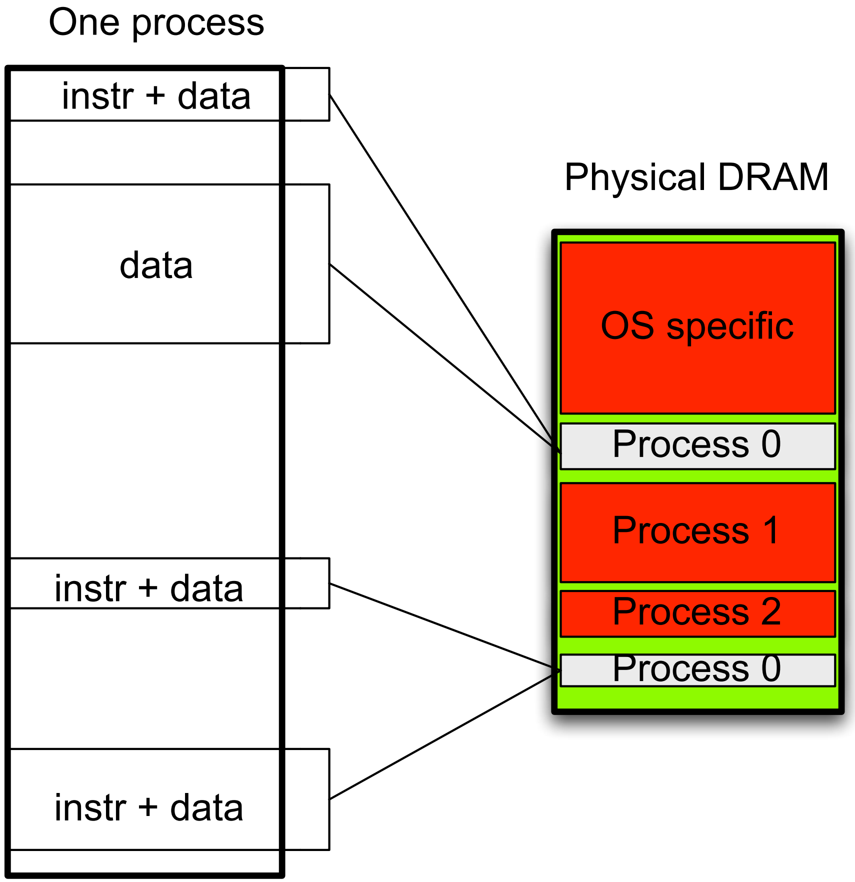
Processes and Memory

- Memory of a process is divided into several parts
- A process can potentially have more memory than system supports
  - Large virtual memory
- Size of a new process' virtual memory address space varies with OS



Processes and Memory

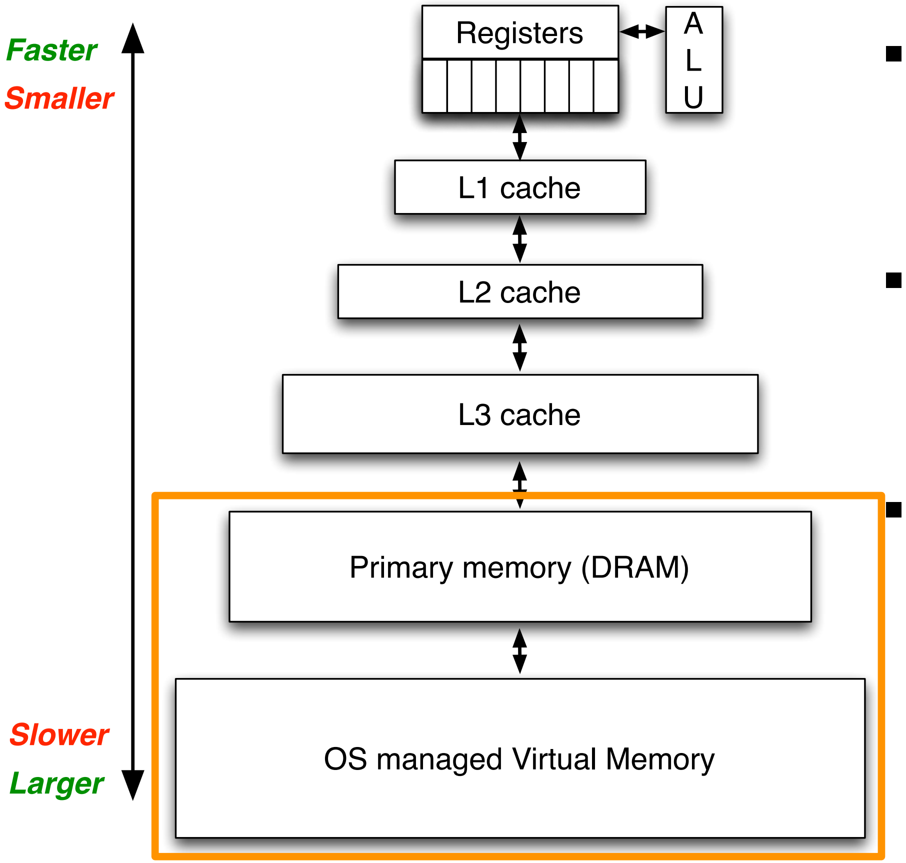
- Virtual memory of a process is mapped to physical memory.
- Physical memory is a mix: cache, RAM, disk, CD, tape, network…



- Multiple processes share the same finite memory resources
  - The physical primary memory (DRAM/SRAM) is easily exhausted
- Whatever cannot fit is stored in secondary memory
  - OS does this management of virtual memory translation to physical memory (with some hardware help)

Processes and Memory

- Process memory exists at different levels
  - based on the need to have instructions and the data ready to continue the program



- OS does most of this work in software
- Memory moves to closest part of CPU for computation
- Stays closer with higher frequency access

Memory: machine level

- Deciding when memory moves up and down the hierarchy *above* primary memory is up to the cache protocol

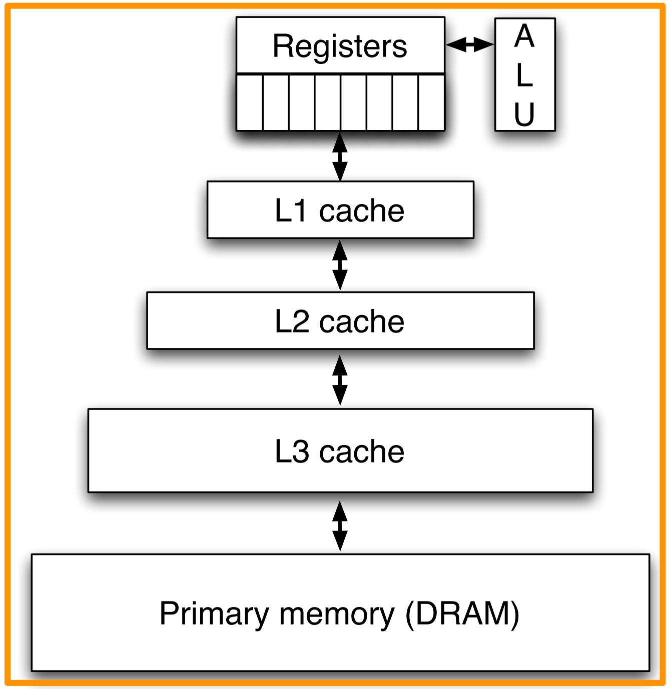
- Software is too slow to manage memory at this level (fine grained)

- Must be done in hardware

- Why?

  Software needs memory

  It is less useful to use more memory to solve memory shortage problems!



Memory: relative costs

- Data that is too large or less frequently used will have to move up/ down the memory hierarchy

- Latency for cache hits and misses access times (Intel):

To Where	Cycles (P2 350MHz)	Cycles (Pentium M)	Cycles (i7 Haswell)
Register	~ 3	<= 1	0
L1 data	~ 6	~ 3	~ 4
L2	~ 57	~ 14	~ 10
L3			~ 40 - 75
Main memory (DRAM)	~ 169 DDR	~ 240 DDR2	~ 200

Memory: relative costs

- L2 cache hit costs are non-uniform

To Where	Intel Core^{TM} 2	i7 Haswell
L1 miss	~ 10	~ 6
L1 Data miss (from L2)	~ 20	~ 10
L2 Hit	~ 12	
L2 Miss	~ 165 desktop / ~ 300 server	~ 50 (from L3)

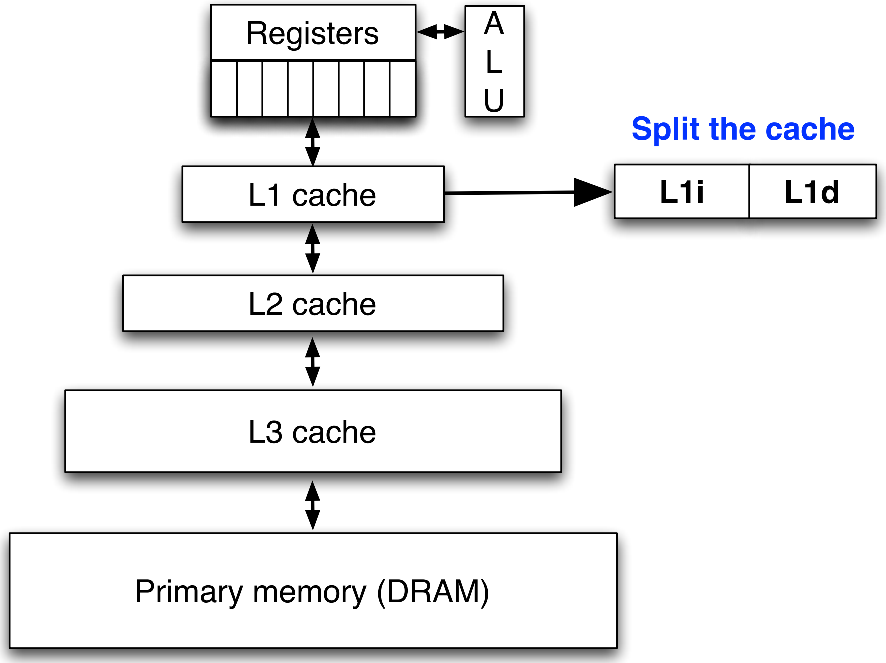
Memory: relative costs

- L3 cache becoming increasingly complex to measure performance

To Where	Intel Core^{TM} i7 and Intel Xeon 5500
L3 cache hit, line unshared	~ 40
L3 cache hit shared line in another core	~ 65
L3 cache hit, modified in another core	~ 75
Remote L3 CACHE	~ 100 - 300

Memory: L1 cache

- Instruction cache – is specifically for storing instructions that have been used and/or will follow
- If a pipeline stall occurs when fetching the next instruction this is catastrophic for performance
  - Intel has employed a separated L1 instruction cache in their CPUs since 1993



Memory: Instruction cache

- Difference of instruction cache to data cache
  1. Amount of instructions to execute is proportional to the size of code
  2. The size of code is proportional to the complexity of problem
  3. Complexity of problem is fixed (SMC exception)

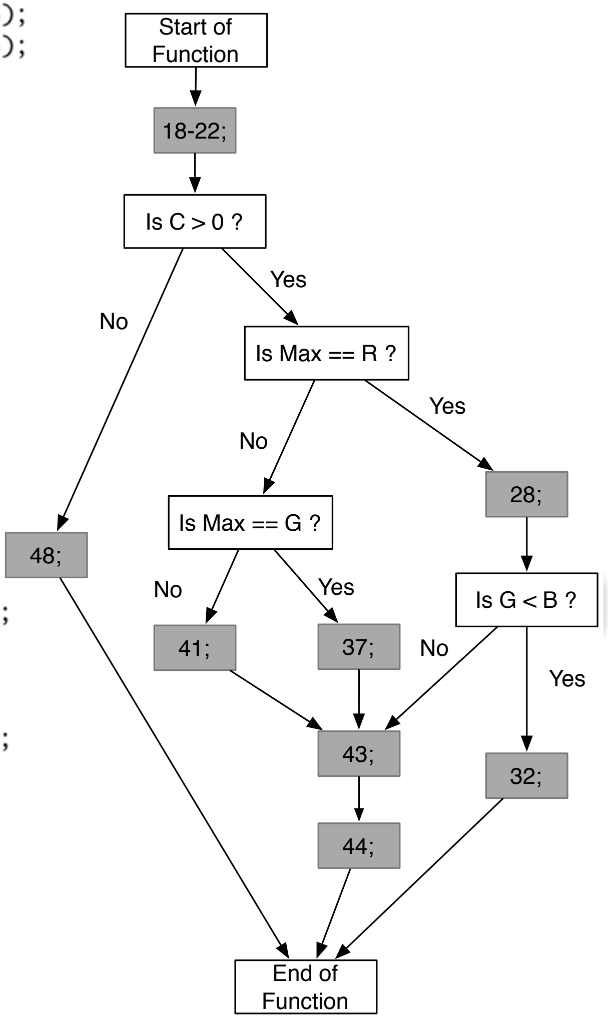
- Programmers design the data handling, but don’t have to think about instructions
  - Compiler writers have good rules for optimising code generation

- Access pattern of instructions much more predictable than data
  - Good in both spatial and temporal locality
  - Don’t need same hardware cache protocol as data, simpler is better!

Memory: instruction access pattern

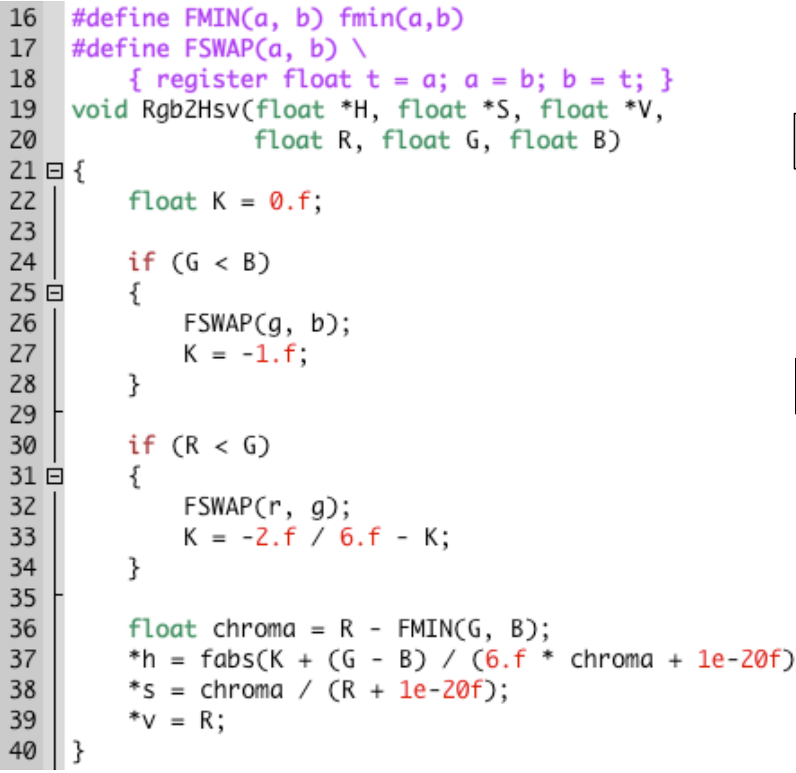
- Convert colour space
  - Red Green Blue -> Hue Saturation Value
- More branches make it harder to predict next instruction to load
- Poor utilisation of prefetch for instructions and memory
- Q: How many unique execution paths?

```
16  void Rgb2Hsv(float *H, float *S, float *V, float R, float G, float B)
17  {
18     float Max = MAX3(R, G, B);
19     float Min = MIN3(R, G, B);
20     float C = Max - Min;
21
22     *V = Max;
23
24     if (C > 0)
25     {
26        if (Max == R)
27        {
28           *H = (G - B) / C;
29
30           if (G < B)
31           {
32              *H += 6;
33           }
34        }
35        else if (Max == G)
36        {
37           *H = 2 + (B - R) / C;
38        }
39        else
40        {
41           *H = 4 + (R - G) / C;
42        }
43        *H *= 60;
44        *S = C / Max;
45     }
46     else
47     {
48        *H = *S = 0;
49     }
50  }
```

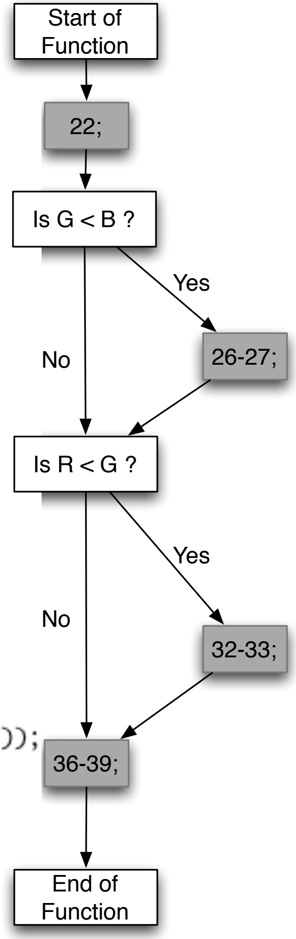


Memory: instruction access pattern

- Convert colour space
- Better version enjoys 10 - 30% increased performance!



```
16  #define FMIN(a, b) fmin(a,b)
17  #define FSWAP(a, b) \
18      { register float t = a; a = b; b = t; }
19  void Rgb2Hsv(float *H, float *S, float *V,
20                 float R, float G, float B)
21  {
22      float K = 0.f;
23
24      if (G < B)
25      {
26          FSWAP(g, b);
27          K = -1.f;
28      }
29
30      if (R < G)
31      {
32          FSWAP(r, g);
33          K = -2.f / 6.f - K;
34      }
35
36      float chroma = R - FMIN(G, B);
37      *h = fabs(K + (G - B) / (6.f * chroma + 1e-20f));
38      *s = chroma / (R + 1e-20f);
39      *v = R;
40  }
```



Memory: data access pattern

- Sequential access

- Depends on what you want to do
  - E.g. Matrix vector multiply on CPU
- Depends on architecture
  - One will work better than another
- Depends on cost of duplication
  - maintain associative data structure
  - memory and computation overhead

- Known limits should be exploited
  - Cache line size (L1 and L2)
  - Cache capacity (keep shared data close)
  - SIMD instructions

```
// Array of structures
struct Vertex { float x, y, z; }
Vertex myvertices[1000];

// structure of arrays
struct VertexChunk
{ float x[1000], y[1000], z[1000]; }
VertexChunk myvertices;

// hybrid, SIMD exploit (4 at a time)
struct Vertex4
{ float x[4], y[4], z[4]; }
Vertex4 myvertices[250];
```

Memory: data access pattern

- Random access
- Linked list and tree like structures
  - Memory is not laid out sequentially
  - Pointer redirection can cause delayed access to data

```
struct _node {
      // other things
      struct _node *next;
}
void list_op(struct _node *head)
{
      struct _node *p;
      p = head;
      while (p->p_next) {
            p = p->p_next;
      }
}
```

- Cache miss: CPU can wait ~100 times longer than it takes to follow the pointer (if the cache has that address)
- Compiler designers try to predict the patterns of data access by code analysis, but if they can't, this makes the purpose of cache for locality of data almost useless
- Just in time compilation can help (Java)
- Allocation of memory*

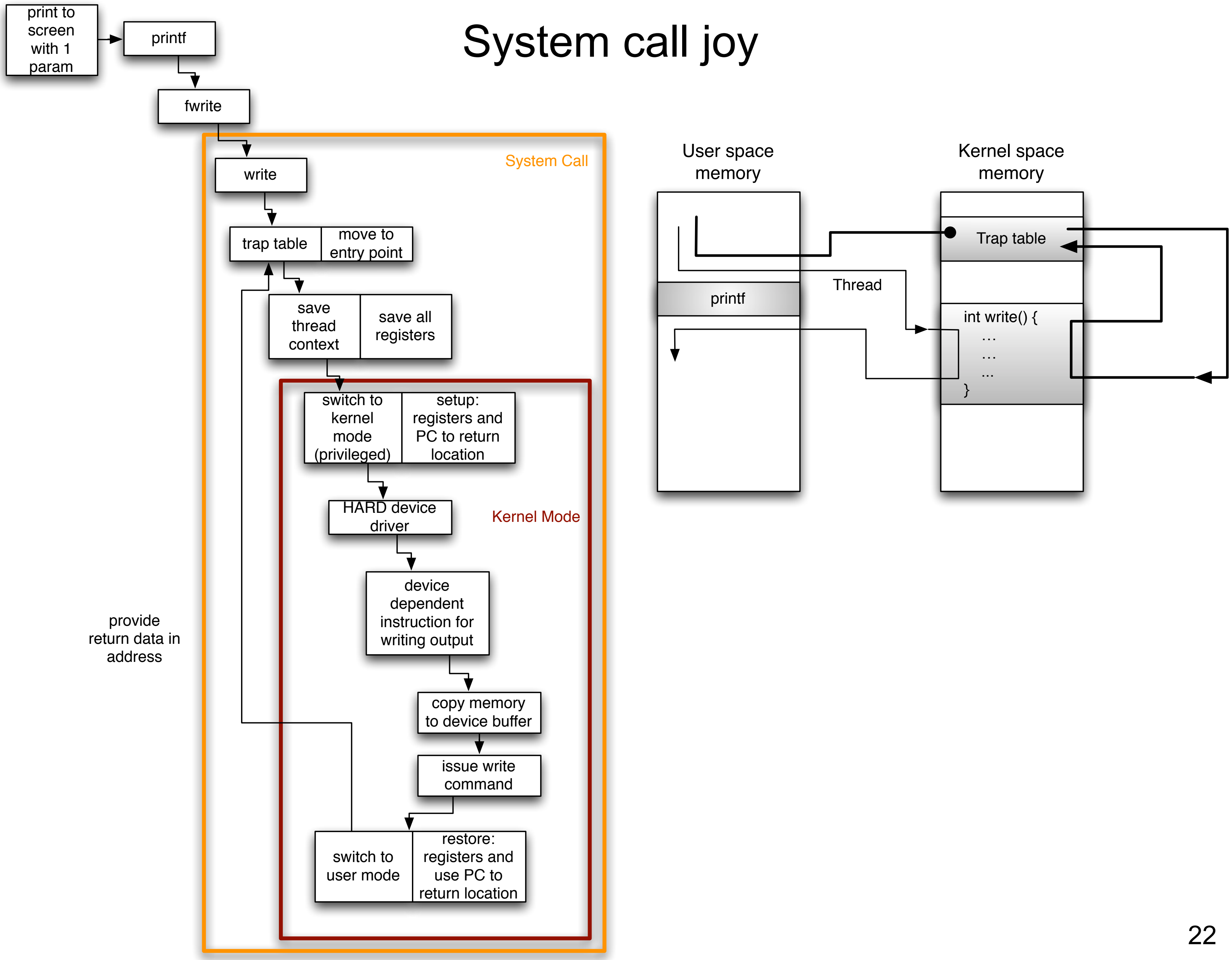
Memory: Exercise

```
void TransformVectors0( float *pDestVectors,
     float const (*pMatrix)[3],
     float const *pSourceVectors, int NumberOfVectors )
{
     int Counter, i, j;
     for ( Counter = 0; Counter < NumberOfVectors; Counter++) {
          for ( i = 0; i < 3; i++ ) {
               float Value = 0.0f;
               for ( j = 0; j < 3; j++ ) {
                    Value += pMatrix[i][j] * pSourceVectors[j];
               }
               *pDestVectors++ = Value;
          }
          pSourceVectors += 3;
     }
}
```

- Convert the above to improve performance

- More techniques for data access to consider
  - Chapter 5: Assigning work to processes statically (0321549422)
    - Block cyclic allocation

System call joy



print to screen with 1 param
printf
fwrite
System Call
write
trap table
move to entry point
save thread context
save all registers
switch to kernel mode (privileged)
setup: registers and PC to return location
HARD device driver
Kernel Mode
device dependent instruction for writing output
copy memory to device buffer
issue write command
switch to user mode
restore: registers and use PC to return location
provide return data in address
User space memory
Kernel space memory
Trap table
Thread
printf
int write() {
…
…
…
}

Outline

- Processes and memory ✓
  - Hierarchy ✓
  - Memory access patterns ✓
- The thread pool

References:
Dr. David Levinthal PhD. Intel Corp. Cycle Accounting Analysis on Intel® Core^{TM}2 Processors.
Dr. David Levinthal PhD. Intel Corp. Performance Analysis Guide for Intel® Core^{TM} i7 Processor and Intel Xeon 5500 Processors. 2009
Ulrich Drepper Red Hat, Inc. What Every Programmer Should Know About Memory. 2007
Chris Hecker Definition Six Inc. PowerPC Compilers: Still Not So Hot. Game Developer June/July 1996
Calvin Lin and Larry Snyder. Principles of Parallel Programming. ISBN 0321549422
Randal E. Bryant and David R. O`Hallaron. Computer Systems: A Programmer`s Perspective 3rd Edition.

Implementing with threads

- Creating and joining threads takes time
  - Pre-create all threads at beginning?
- Task parallelism: e.g. threads for specific functions foo(), bar()
  - Will all tasks finish the same time?
  - Idle threads waiting on data for a specific task could be more useful
  - Tasks may have equal weight, but data can vary also
- Data parallelism
  - what assumptions about available threads have you made?
- What is the problem relationship with threaded programming
  - Independent or cooperative work?

Thread pool

- A thread pool is a collection of **N** threads that will perform a general computation task.

- Objectives
  - amortise time cost by load balancing smaller workloads
  - an opaque parallel programming construct to aid non-parallel programmer
    - Reliability of synchronisation/performance
    - For good software construction: definition of work (Object Oriented Design)
  - Optimisation possible with other parts of system by varying **N** threads (JVM)

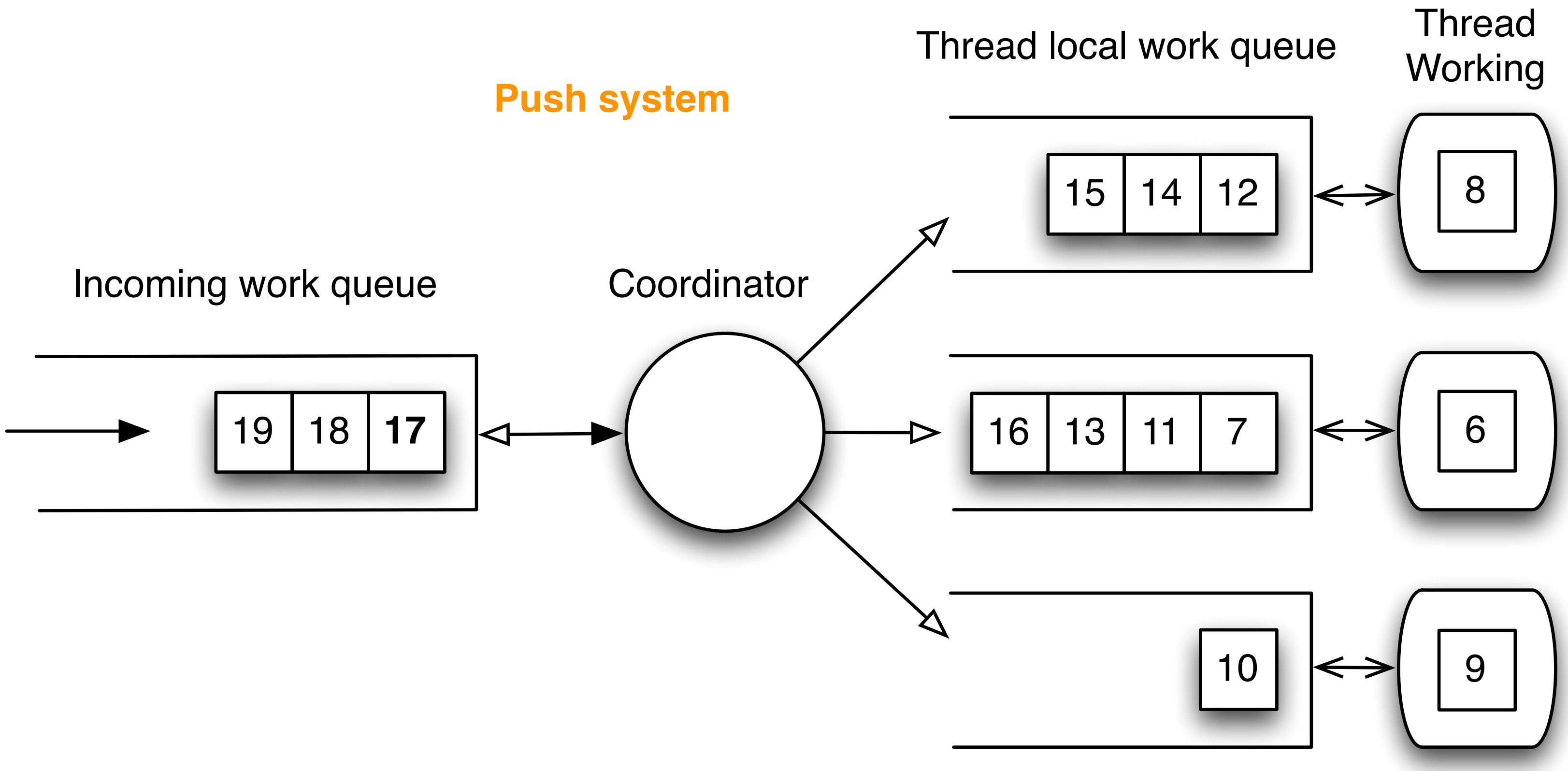
Thread pool

- Pthread already allows us to define a general computation task
- **void *func(void *arg);**
- Any function we define as this prototype can be passed around by using it’s function pointer

```
struct frame_info {
  uint8_t  magic[2];
  uint8_t  control;
   …
  uint32_t time_stamp;
  uint8_t  data[];
 };

void kinect_1stpass(struct frame_info *fm);

// wrap existing function
void *
kinect_1stpass_for_pthreadpool(void
*param)
{
       // extract parameters
       struct frame_info *fm =
                  (struct frame_info*) param;

       kinect_1stpass(fm);

       return NULL;
}
```

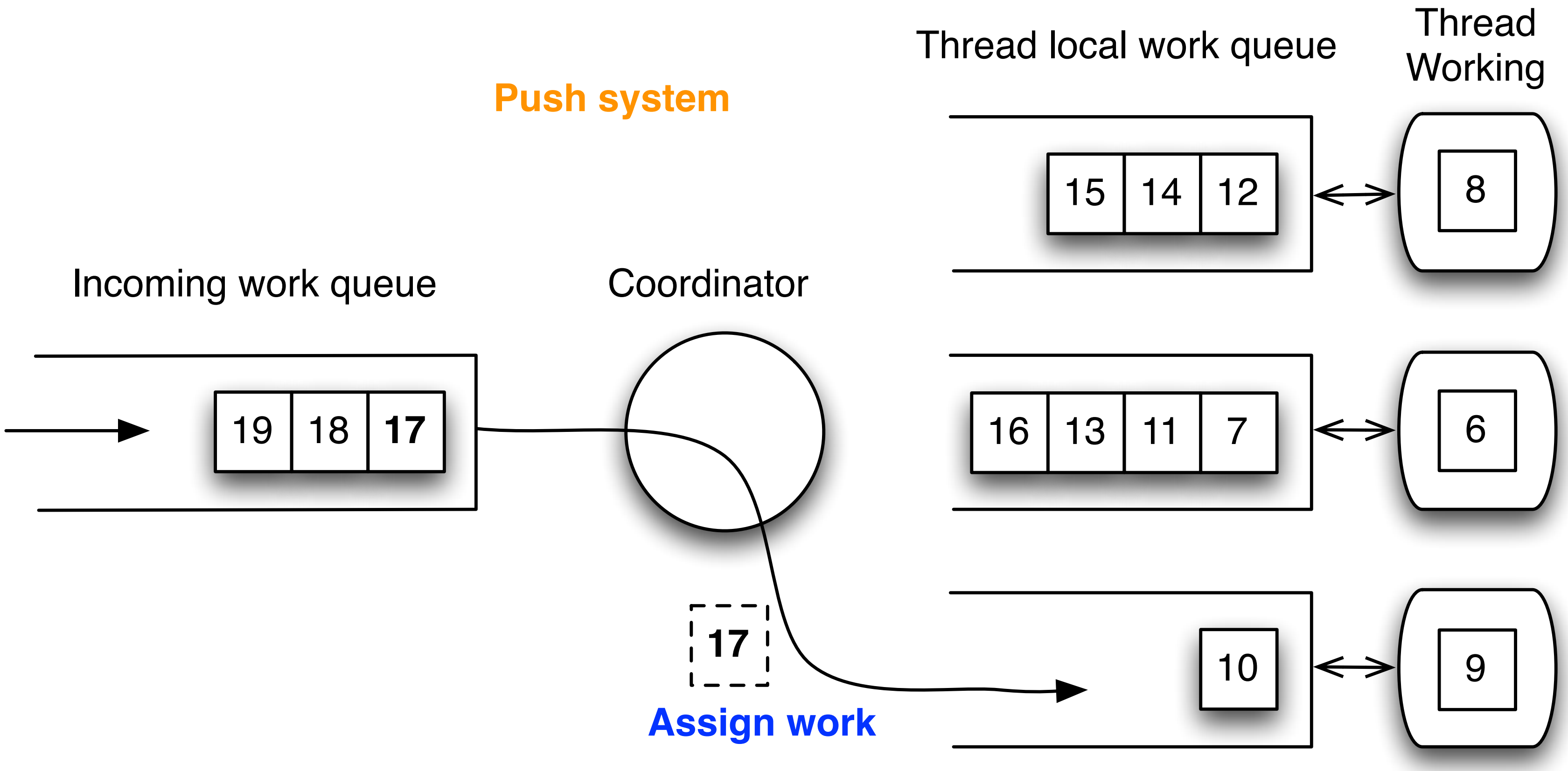
Thread pool

- How is the work distributed among N threads?



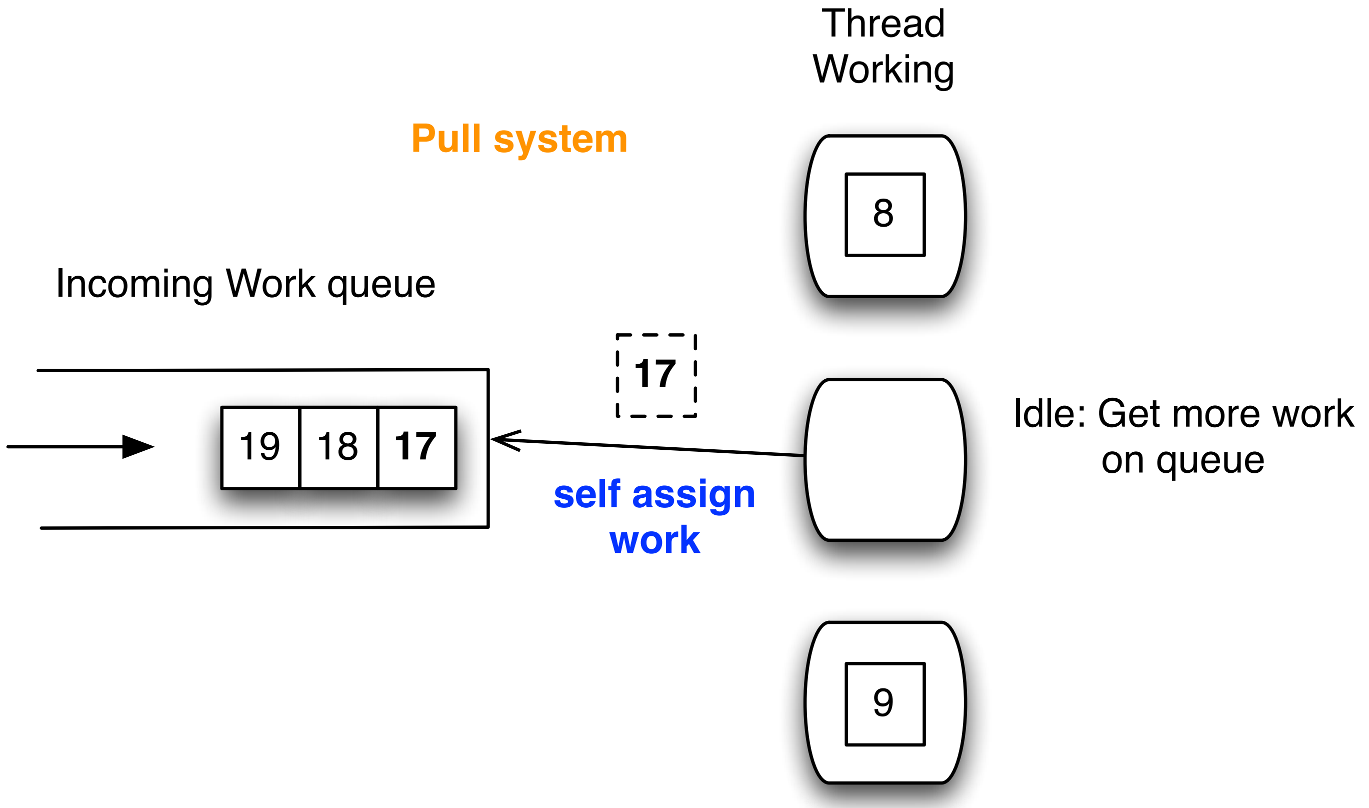
Thread pool: Push work to threads

- Check work load of each thread worker
  - Separate thread for coordinator?



Thread pool: Pull from work queue

- Don’t need private thread queue
  - Consequences?



Thread pool: build your own

- A structure to define the general computation
- Functions for the User of the threadpool
  - Creating/Destroying
  - Blocking/non-blocking?
- User synchronisation
  - Thread pool is meant to disassociate work and thread but…
  - Results are still needed
  - Multiple work items stem from larger sets of tasks

```
// basic unit of work
struct _workitem {
        void *(*action)(void*);
        void *arg;
};
typedef struct _workitem workitem_t;

extern
threadpool_t *
luthreadpool_init(
        int max_threads );

extern
void
luthreadpool_q_workitem(
        threadpool_t *tp,
        void *(*action)(void*),
        void *arg );

extern
void
luthreadpool_barrier(
        threadpool_t *tp,
        workitem_t *witem,
        int witem_count );
```

Thread pool: build your own

- Structure to track each thread worker
- The general worker function
- Communication within the thread pool
  - How to synchronise getting work?
  - What to do after finished work?
- Fairness

```
luthreadpool_init(…)
{
  pthread_attr_init(&attr);
  pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
  pthread_attr_setschedpolicy(&attr, SCHED_RR);

  for ( ; i < max_threads; ++i)
    pthread_create(&tp->threads[i], &attr, luthreadpool_start, wd);
}
```

```
// per thread data
struct _worker_data {
        int id;
        …
        struct threadpool_t *tp;
};
typedef struct _worker_data workerdata_t;

static
void *
luthreadpool_start(void *arg)
{
        // work forever
        while (1) {

                // get workitem
                workitem_t *work = …

                // do the general computation
                (*work->action)(work->arg);

                // anything else?
                free(work);
        }
}
```

Thread pool: design considerations

- Task size: meant to be small, but how small?
  - Experimentally, find the threshold on unloaded machine
  - Dynamically, do a small test in the program! (beware cold/hot cache)
- Overhead costs
  - additional function calls
  - reformatting parameters to be described with one address (pthread)
  - Inside threadpool there are delays which has nothing to do with task
- Which data structure would minimise contention for distributing work?
- < 200 lines of C code (using pthreads)