Untangling Threads

Introduction

Brett Searles

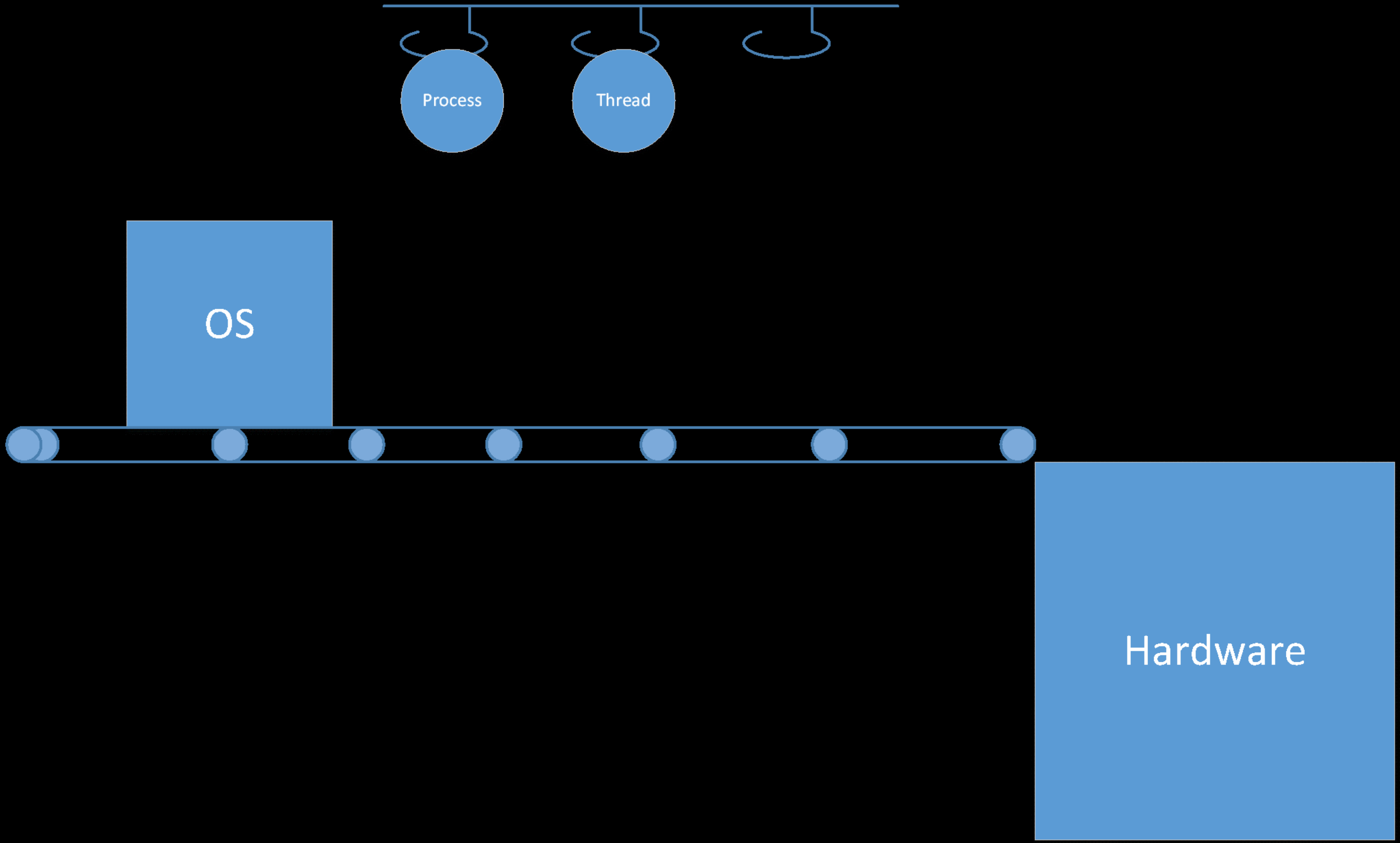
Principle Architect / Developer

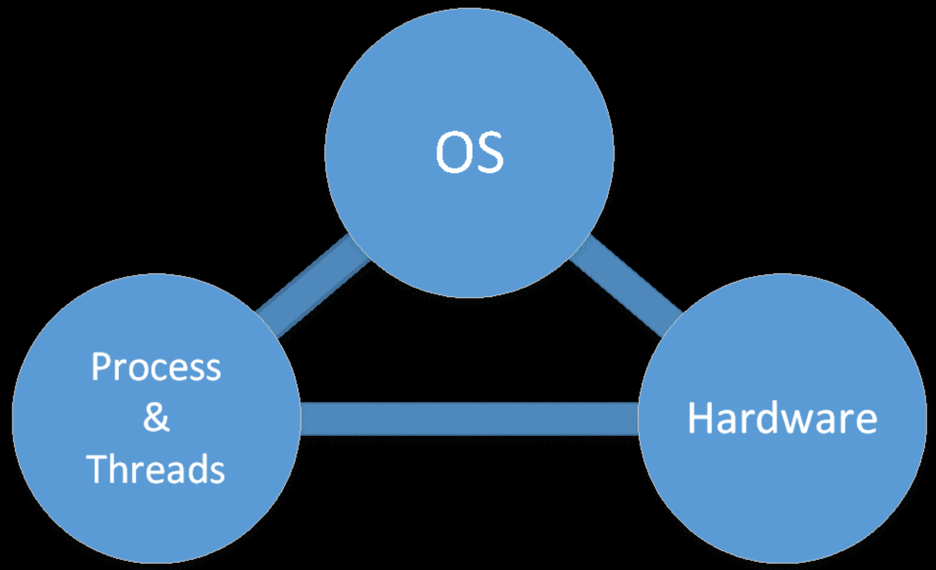
Attobotics

brett.searles@attobotics.net

Introduction

- To examine all of the components that affect thread management
- To explore what exists already to better utilize
- Create STL container
  - Interacts with the OS and Hardware
  - This interaction is used to control thread synchronization
  - Work on various hardware and memory configurations
  - Improve efficiency in processing multiple threads


Process
Thread
OS
Hardware


OS
Process
&
Threads
Hardware

Prior Art & Current Trends

*nix based C

- Considers threads as lightweight processes
- Creates threads via a pthread_... Method
- Destroys threads via Join or Exit call.
- Can perform stack management; however, still dependent on OS and hardware
- Supports synchronization types
  - Mutex
  - Condition
  - Barrier
  - Reader-Writer
  - Spin locks

C++ Standard

Thread

Mutex

Condition Variables

- wait…(m)
- notify…

Futures

```
class thread {
    public:
        class id;
        typedef implementation-defined native_handle_type;
        // construct/copy/destroy:
        thread() noexcept;
        template <class F, class ...Args> explicit thread(F&& f,
        Args&&... args);
        ~thread();
        thread(const thread&) = delete;
        thread(thread&&) noexcept;
        thread& operator=(const thread&) = delete;
        thread& operator=(thread&&) noexcept;
        // members:
        void swap(thread&) noexcept;
        bool joinable() const noexcept;
        void join();
        void detach();
        id get_id() const noexcept;
        native_handle_type native_handle();
        static unsigned hardware_concurrency() noexcept;
    };
}
```

Concurrency

- Latches
  - Uses internal counter for blocking threads. When counter reaches 0, all blocked threads are released
  - Cannot be reused.
- Barriers
  - Thread coordination mechanism that optimizes memory access.
  - Once the operation is completed, can be reused

Lock Free

- Lock Free [7]
  - Atomic smart pointers
    - Utilizes the hardware to lock with the following commands:
      - ADD, ADC, AND, BTC, BTR, BTS, CMPXCHG, CMPXCH8B, DEC, INC, NEG, NOT, OR, SBB, SUB, XOR, XADD, and XCHG
      - Cache lock rather than a bus lock
- Boost.Lockfree [8]
  - Provides data structures or policies to provide nonblocking performance

Parallel

- Multiple Execution Cores for processing
- Architectures
  - multiple CPU cores
  - SIMD
  - MIMD
  - MISD
  - Etc.
- C++ is developing the language to support the above

Co-Routines

- Basically one process communicates to another process that it “yield”s to the other.
- Possible to send also results back to the thread(s) to process

Design Decisions

Influencers

- Hardware
- Memory
- OS Scheduler
- Process ,i.e., application
- Threads

Hardware

- Platform
  - 32 bit
    - x86
    - ARM
  - 64 bit
    - x64
    - ARM
  - Multi core
  - GPU (Graphics Processing Unit)
  - GPGPU (General Purpose Graphics Processing Unit)
  - FPGA (Field Programmable Gate Array)
  - MIMD (Multiple Instruction Multiple Data)

32 bit

- x86
  - Segments
    - CS, DS, ES, FS, GS
  - Descriptor Tables (DT)
    - Global DT
    - Local DT
- ARM
  - Have different registers
  - R13 for stack
  - Meant more for multi-tasking

x64

- x64
  - Eliminated segments
  - 10^{18} address space

- ARM
  - Added more registers
  - More instructions
  - Still focused on embedded, smartphones and tablets

Execution Units

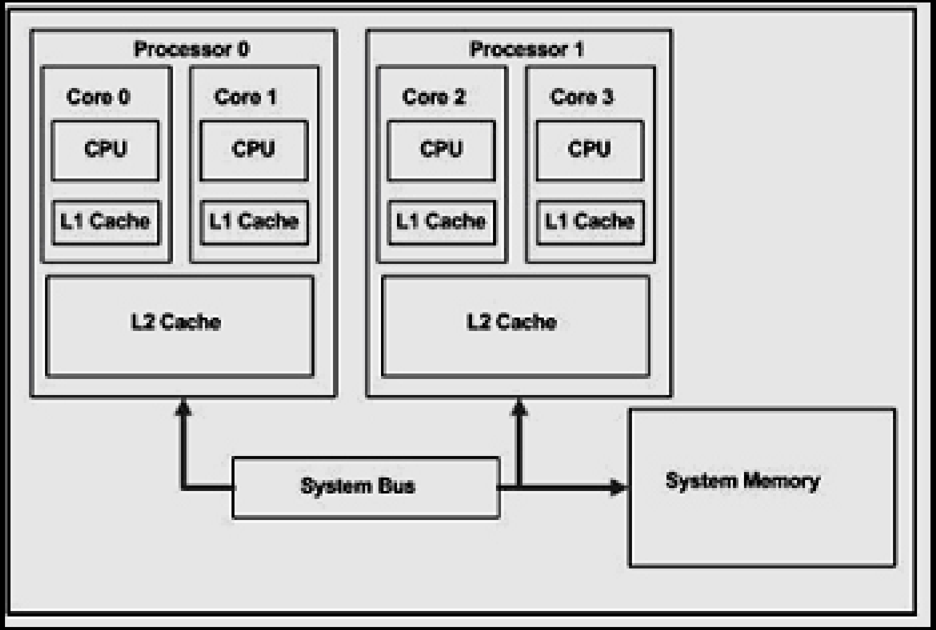
Multicore

May be any architectures with multiple execution units

Need to set affinity for a thread so that it will operate only on one processor

Two ways to supervise operations

1) SMT – Symmetric Multi-Processor
2) AMT – Asymmetic Multi-Processor



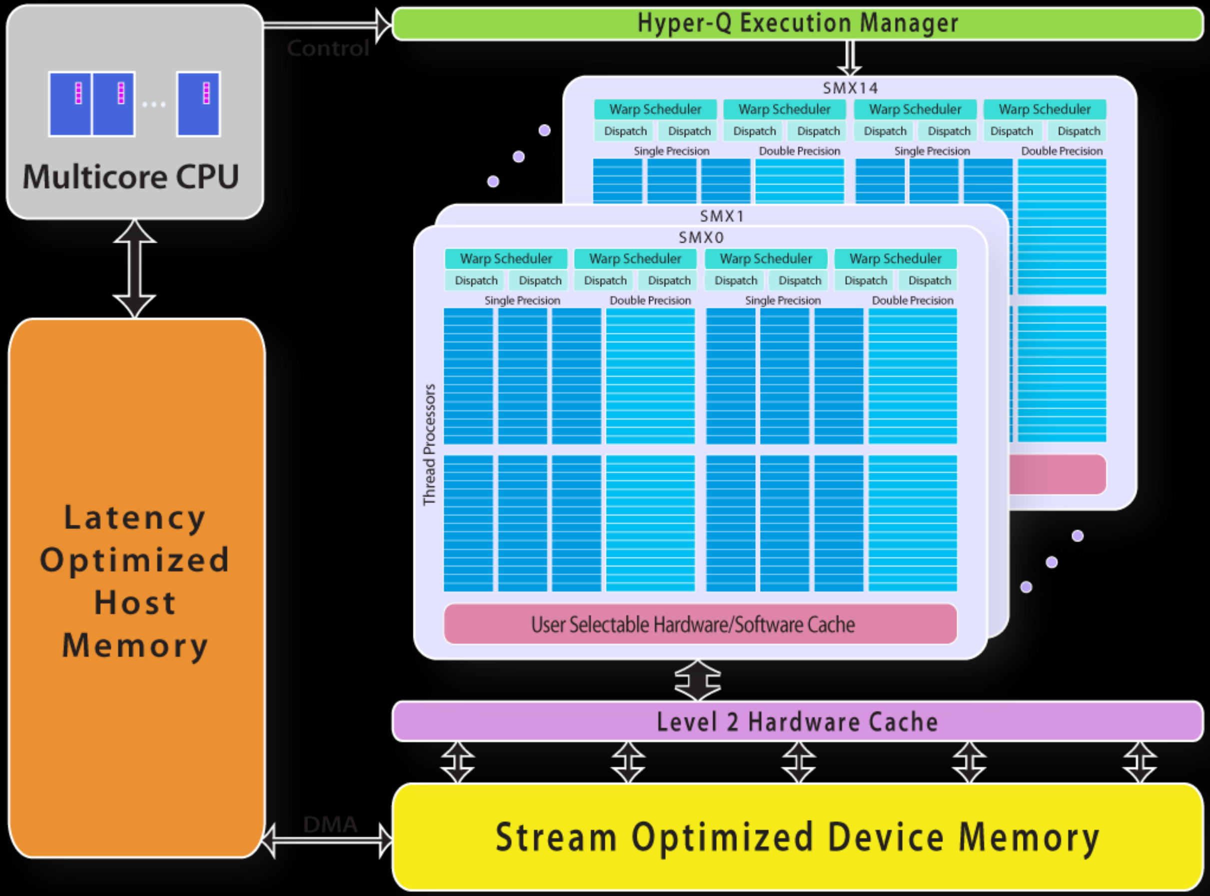
GPU

Highly parallel processing

Specific to basically doing high speed calculations

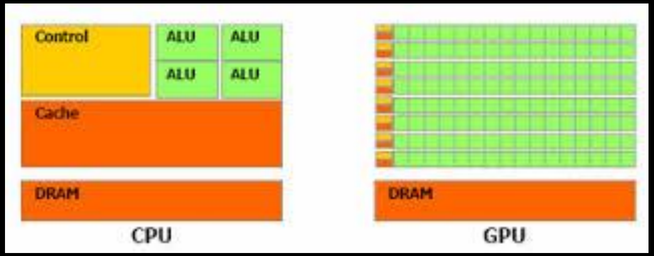
Configured to to apply a single instruction to multiple data (SIMD)

Uses threading, yet don't support interrupts and exception.



GPGPU

Combines the controlling features of the CPU with the processing power of SIMD of a GPU

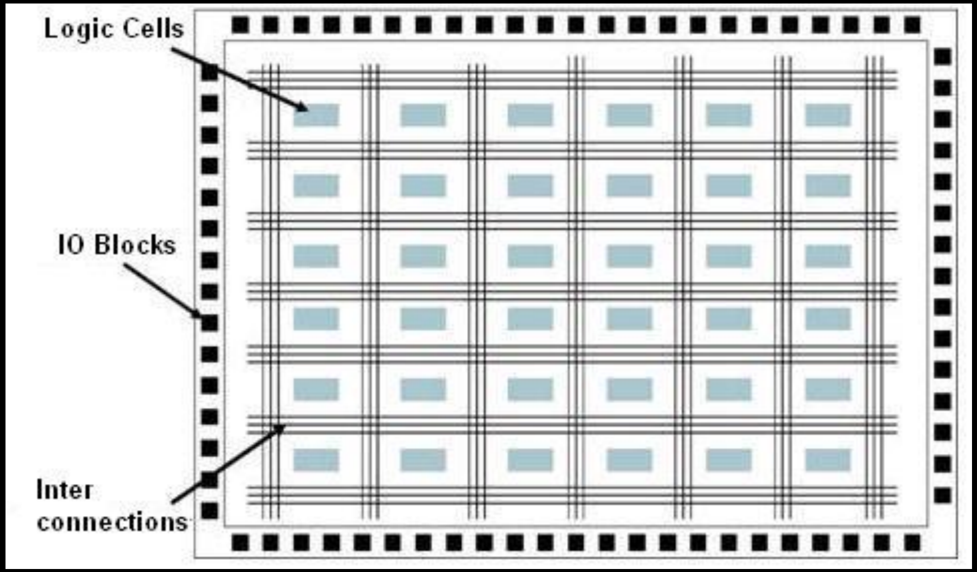


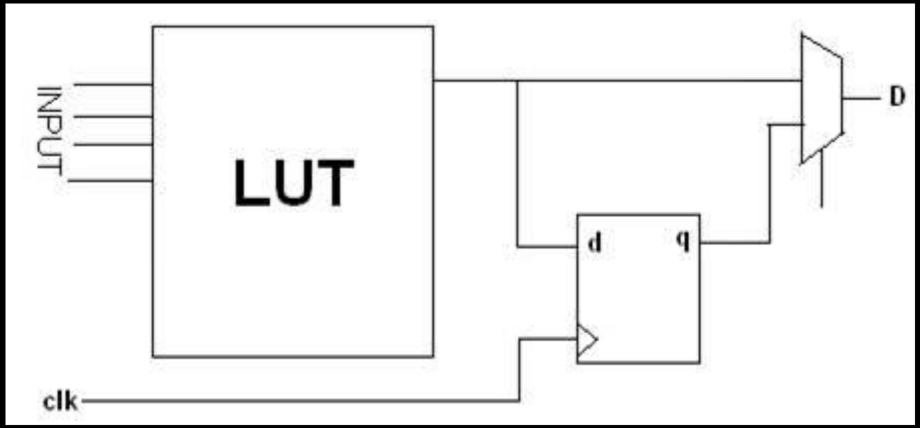
FPGA

Field-Programmable Gate Array

Can place ARM or some other parallel processing cores onto a gate array.

It may work in conjuction with a controller external to the gate array configuration

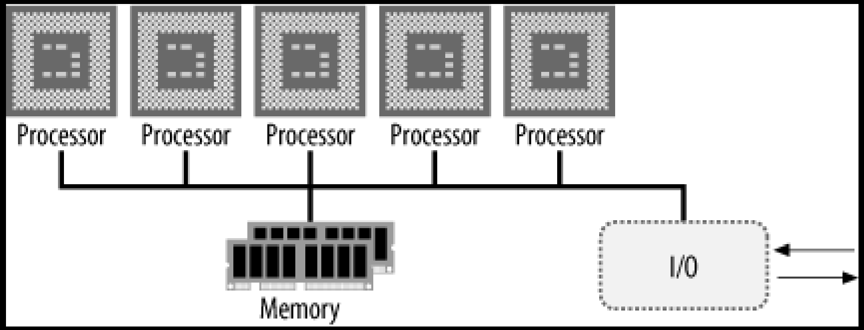




MIMD

Multiple Instruction Multiple Data

Need to highly synchronize data between processors in order to insure correctness of results



Memory

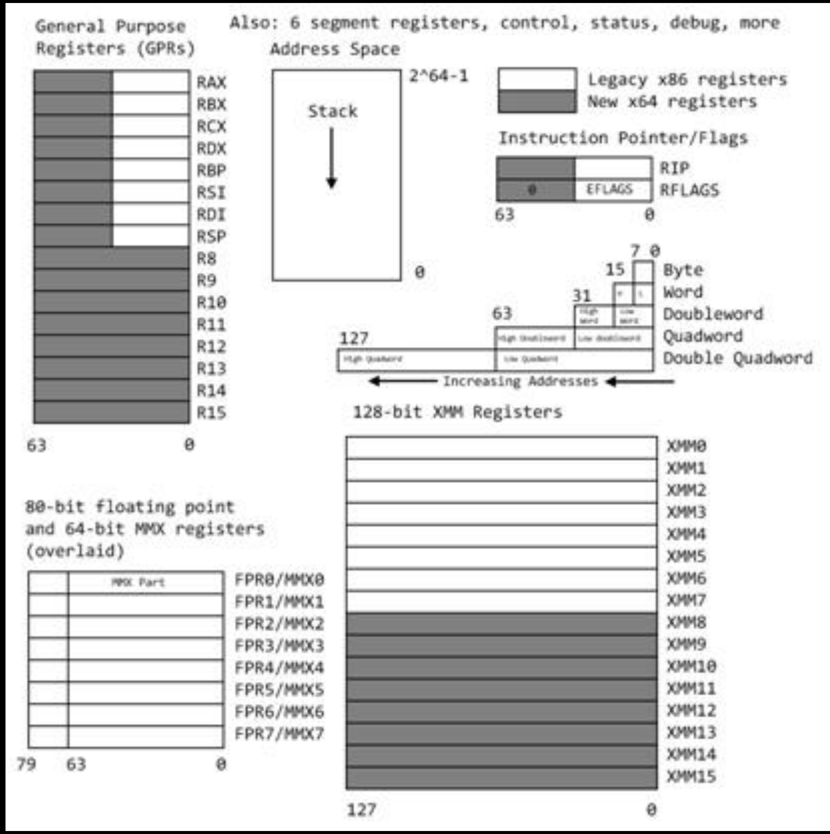
- Registers – How many?
- Cache – How Large?
  - L1
  - L2
  - L3
- RAM – How large and fast?
  - Stack
  - Heap
- GPU configuration – Fast?

Registers

- Memory units that are closest to the execution unit
- Very fast to access
- Yet very limited in number

Intel (CISC based)

- (e)ax, (e)bx, (e)cx, (e)dx, (e)bp, (e)si, (e)di, (e)sp, (e)flags (32)
- r.. (64)
- Mmx… for multi-media
- Xmm… for SIMD
- Fpr… for floating point



ARM (RISC based)

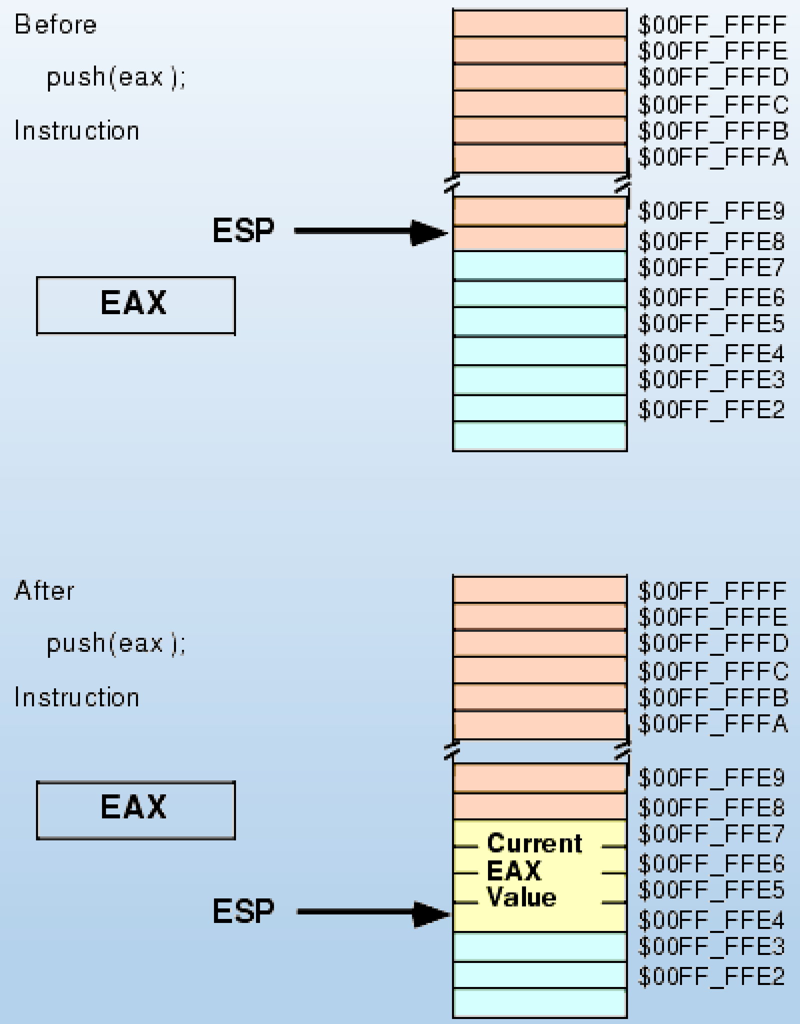
- r1 – r15 for both user and supervisor modes
- Have r13-r14 for monitor, abort, undefined modes and IRQ
- FIQ contains R8, 9, 10, 11, 12, 13, and 14
- The last two points require to be in supervisor mode

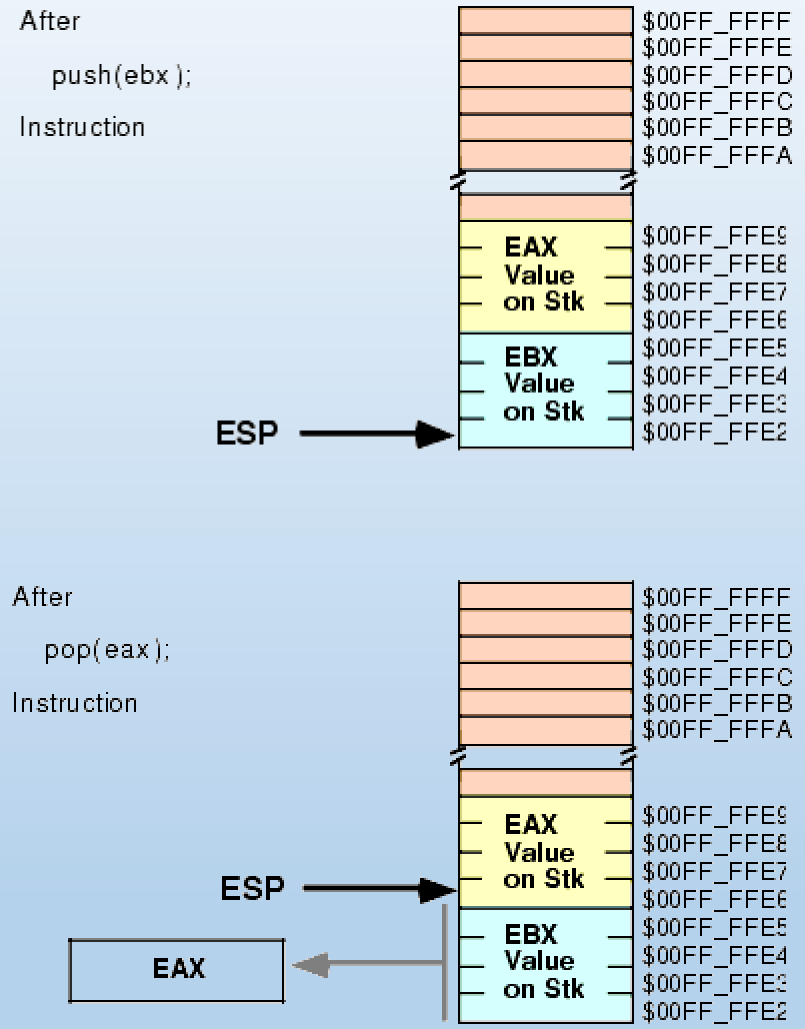
Application level view	System level views							
		Privileged modes						
			Exception modes					
	User mode	System mode	Supervisor mode	Monitor mode ‡	Abort mode	Undefined mode	IRQ mode	FIQ mode
R0	R0_usr							
R1	R1_usr							
R2	R2_usr							
R3	R3_usr							
R4	R4_usr							
R5	R5_usr							
R6	R6_usr							
R7	R7_usr							
R8	R8_usr							R8_fiq
R9	R9_usr							R9_fiq
R10	R10_usr							R10_fiq
R11	R11_usr							R11_fiq
R12	R12_usr							R12_fiq
SP	SP_usr		SP_svc	SP_mon ‡	SP_abt	SP_und	SP_irq	SP_fiq
LR	LR_usr		LR_svc	LR_mon ‡	LR_abt	LR_und	LR_irq	LR_fiq
PC	PC							
APSR	CPSR							
			SPSR_svc	SPSR_mon ‡	SPSR_abt	SPSR_und	SPSR_irq	SPSR_fiq

‡ Monitor mode and the associated banked registers are implemented only as part of the Security Extensions

Stack

- Usually top down (subtracting)
- Each process is allocated a segment of the stack (OS-dependent)
- Each Thread has allocated space
- very fast access
- Variables are deallocated when function or process ends
- space is managed efficiently by CPU

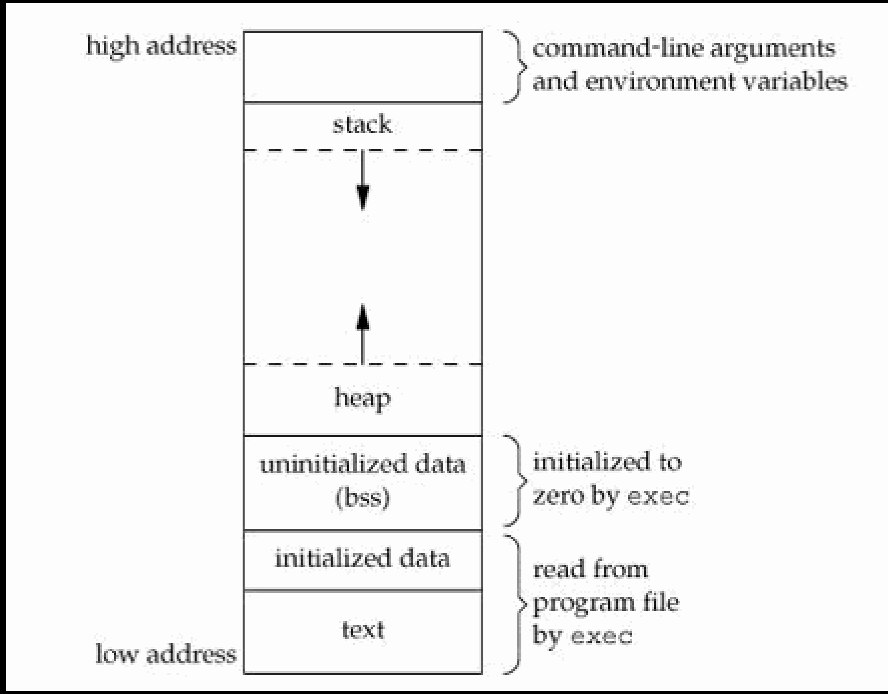




After
push(ebx);
Instruction
$00FF_FFFF
$00FF_FFFE
$00FF_FFFD
$00FF_FFFC
$00FF_FFFB
$00FF_FFFA
EAX Value on Stk
EBX Value on Stk
$00FF_FFE
$00FF_FFE
$00FF_FFE
$00FF_FFE
$00FF_FFE
$00FF_FFE
$00FF_FFE
$00FF_FFE
ESP
After
pop(eax);
Instruction
$00FF_FFFF
$00FF_FFFE
$00FF_FFFD
$00FF_FFFC
$00FF_FFFB
$00FF_FFFA
EAX Value on Stk
EBX Value on Stk
$00FF_FFE
$00FF_FFE
$00FF_FFE
$00FF_FFE
$00FF_FFE
$00FF_FFE
$00FF_FFE
$00FF_FFE
ESP
EAX

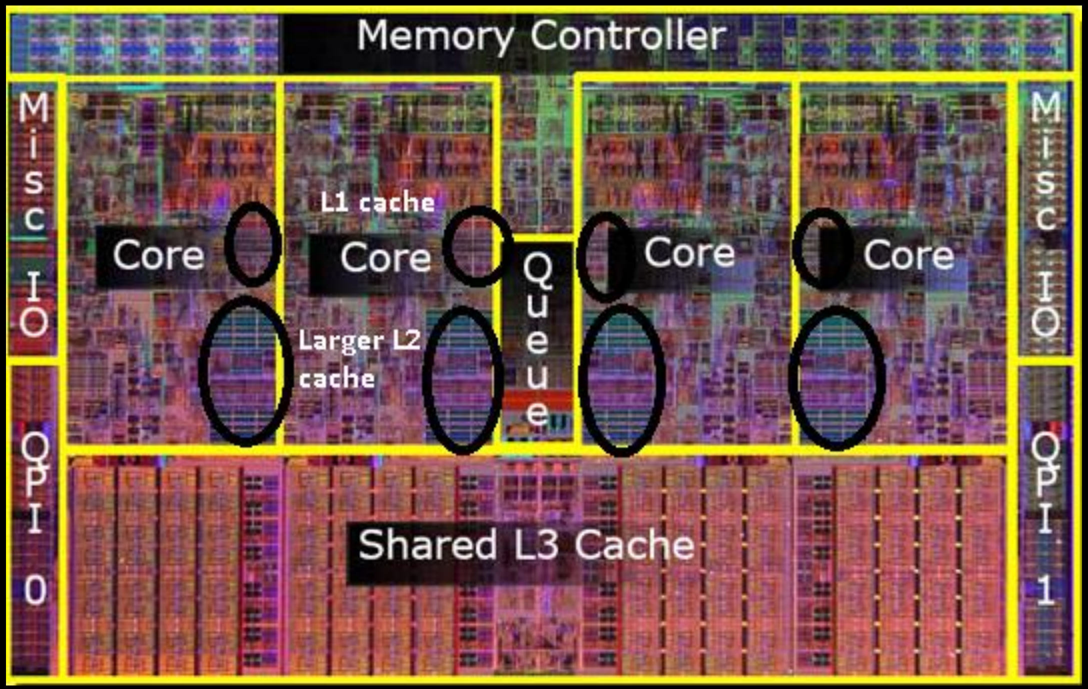
Heap

- Size limitation is size of memory minus the size of stack, code and possibly other variable placeholders.
- May contain large memory for single variables
- Accessible throughout the lifetime of the process
- Need to be explicitly deleted
- Can lead to fragmentation



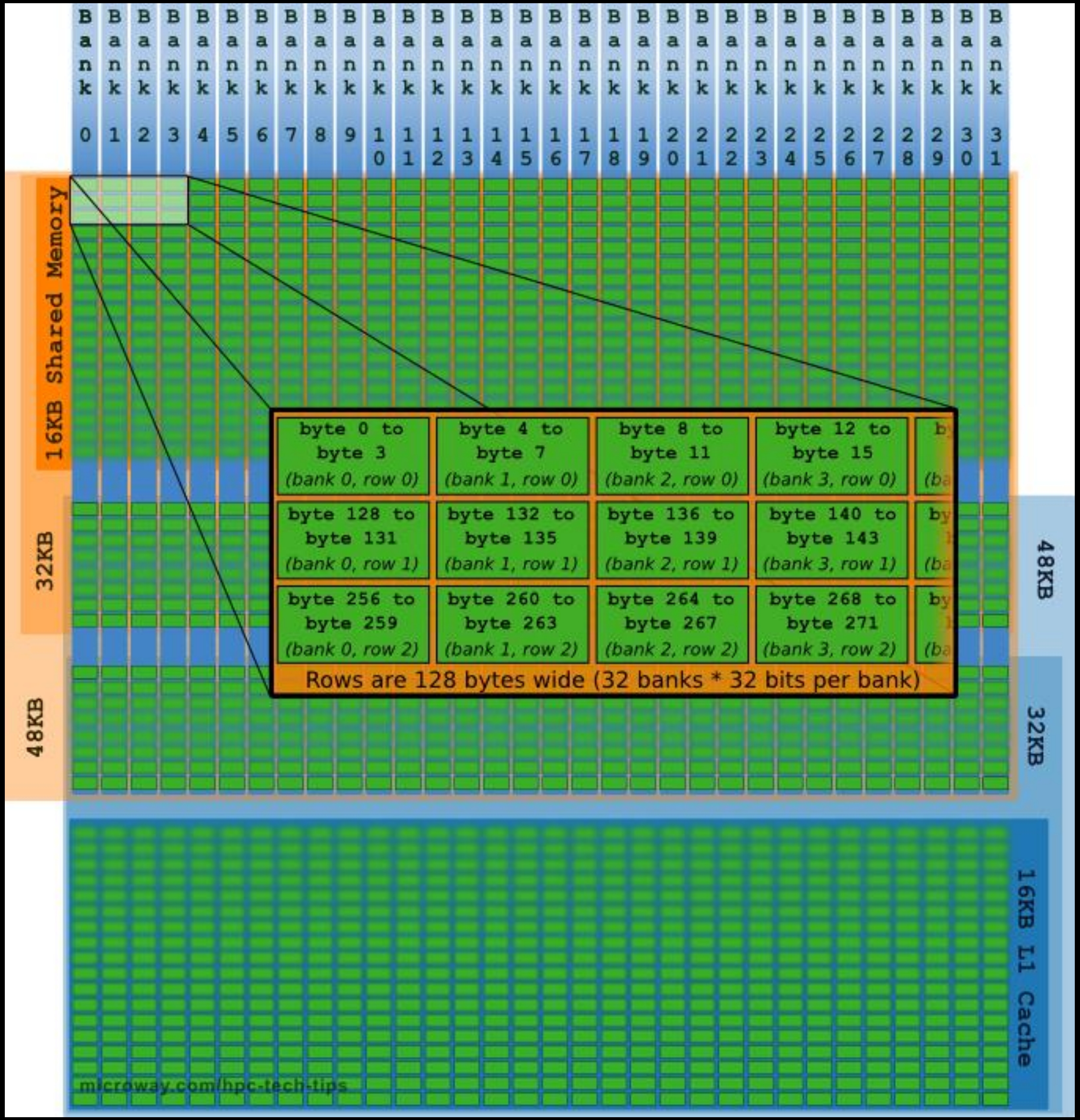
Cache

- L1 – local to Processor core
- L2- shared between two cores on the same die
- L3 – shared between all core
- Uses Read and Write fences to synchronize data flow
- Hit-Miss Ratio



GPU configuration

- Local on chip
  - Register
  - Shared
- Reside Off Chip
  - Local
  - Constant
  - Texture
  - Global



OS Scheduler

- Various algorithms to handle process and thread execution
  - 1st come, 1st serve
  - Round Robin
  - Priority
  - Multi-Level Feedback Queues
  - Lottery
- Want to avoid deadlocks, even livelocks, so most use Preemptive scheduling
- Three types of scheduling
  - Long Term
  - Medium Term
  - Short Term

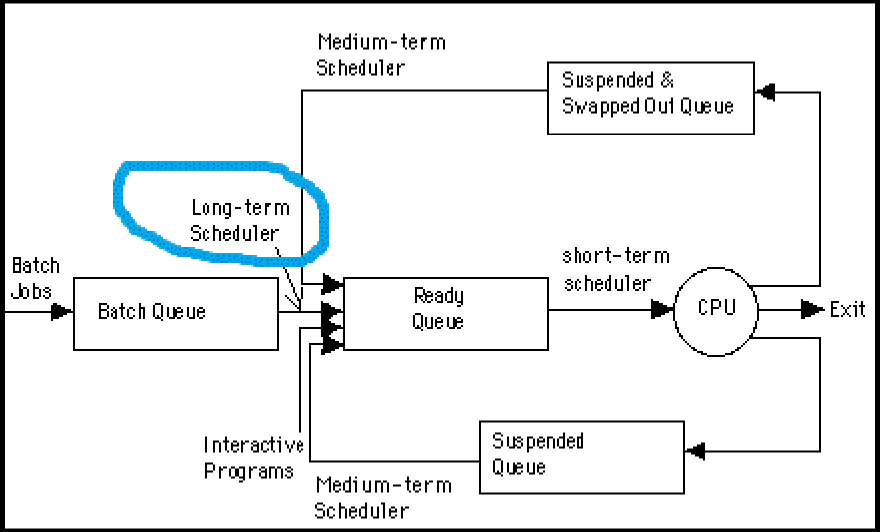
Long

If the OS uses pre-emptive scheduling, the long term scheduler does not exist.

However, if does exists, all new processes will be determined for admittance to the process queue.

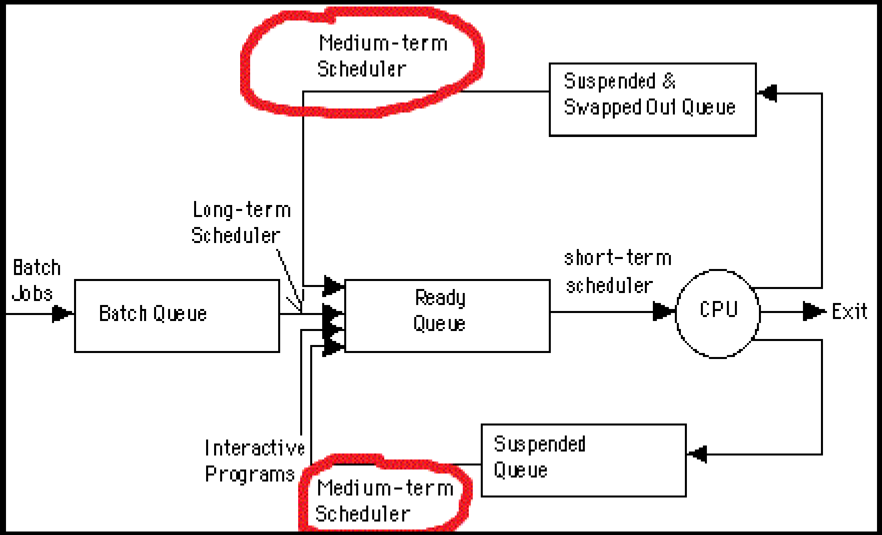
This scheduler balances the jobs between I/O and processor based processes.

Loads the process by allocating memory and setting variables used by the process.



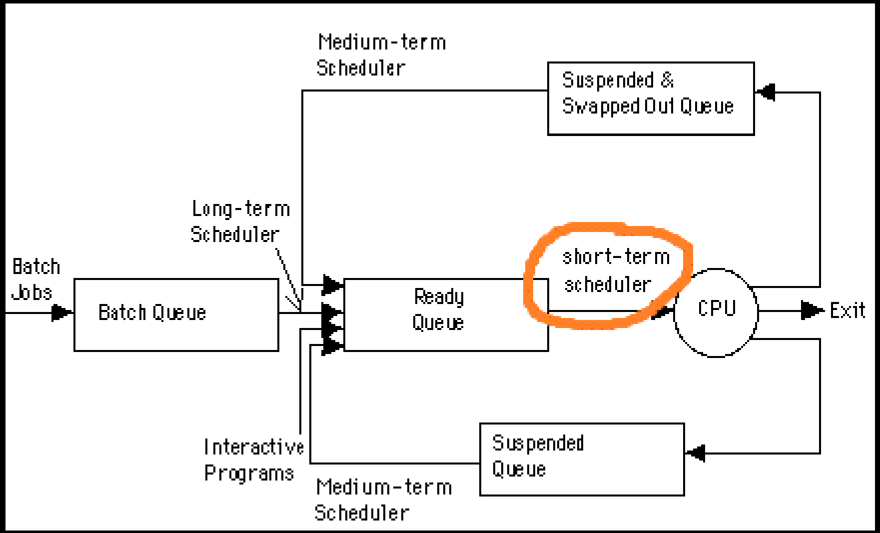
Medium

Removes processes from memory



Short

Used to change the state of the process from "Ready" to "Running"



Execution Scenarios

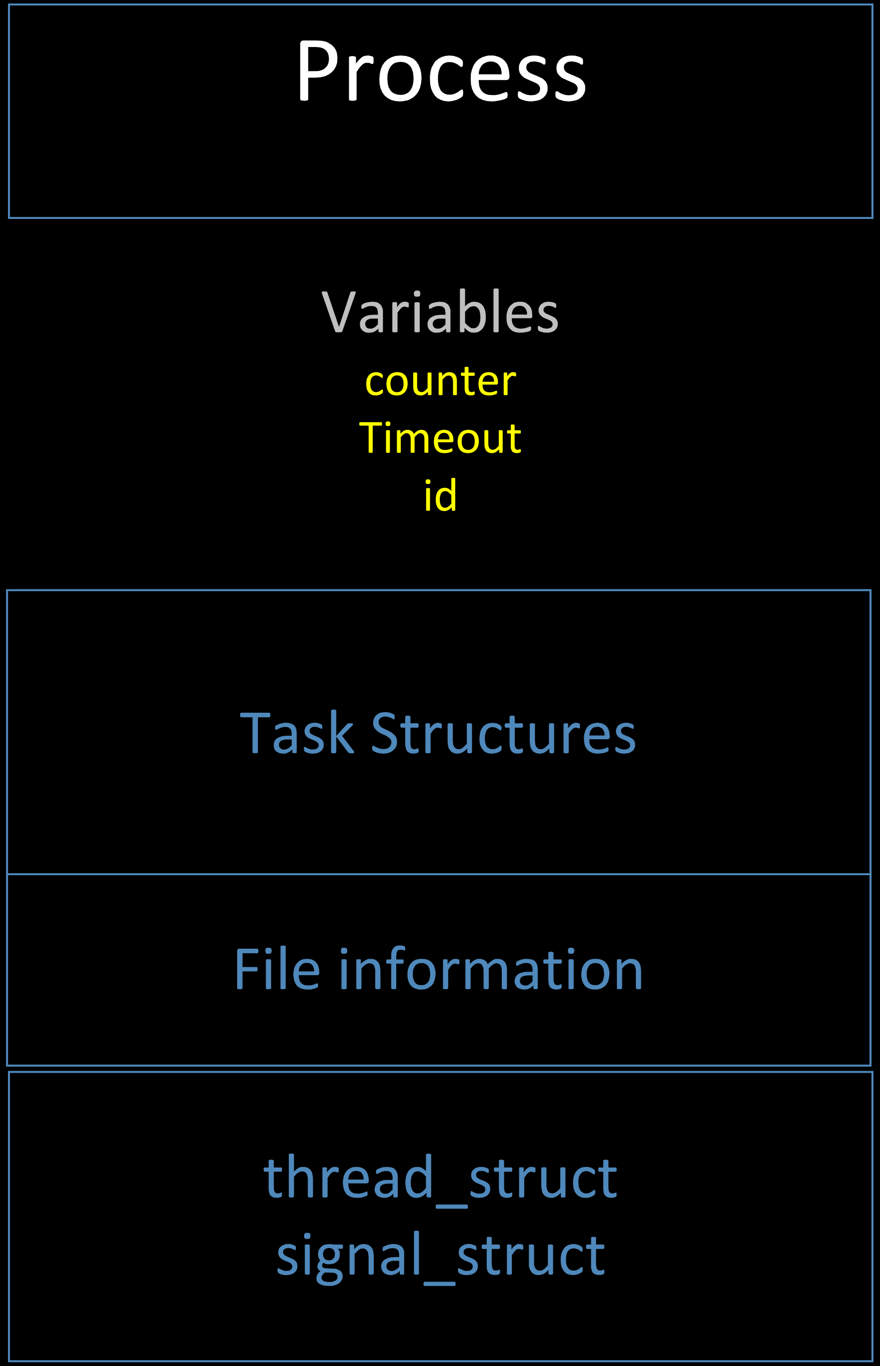
Process

Is the main execution unit for an application

Stores all of the information to successfully operate an application

Stores thread information, interrupts and other information that will be used by the app.

In Linux and Windows, that information is stored in a struct



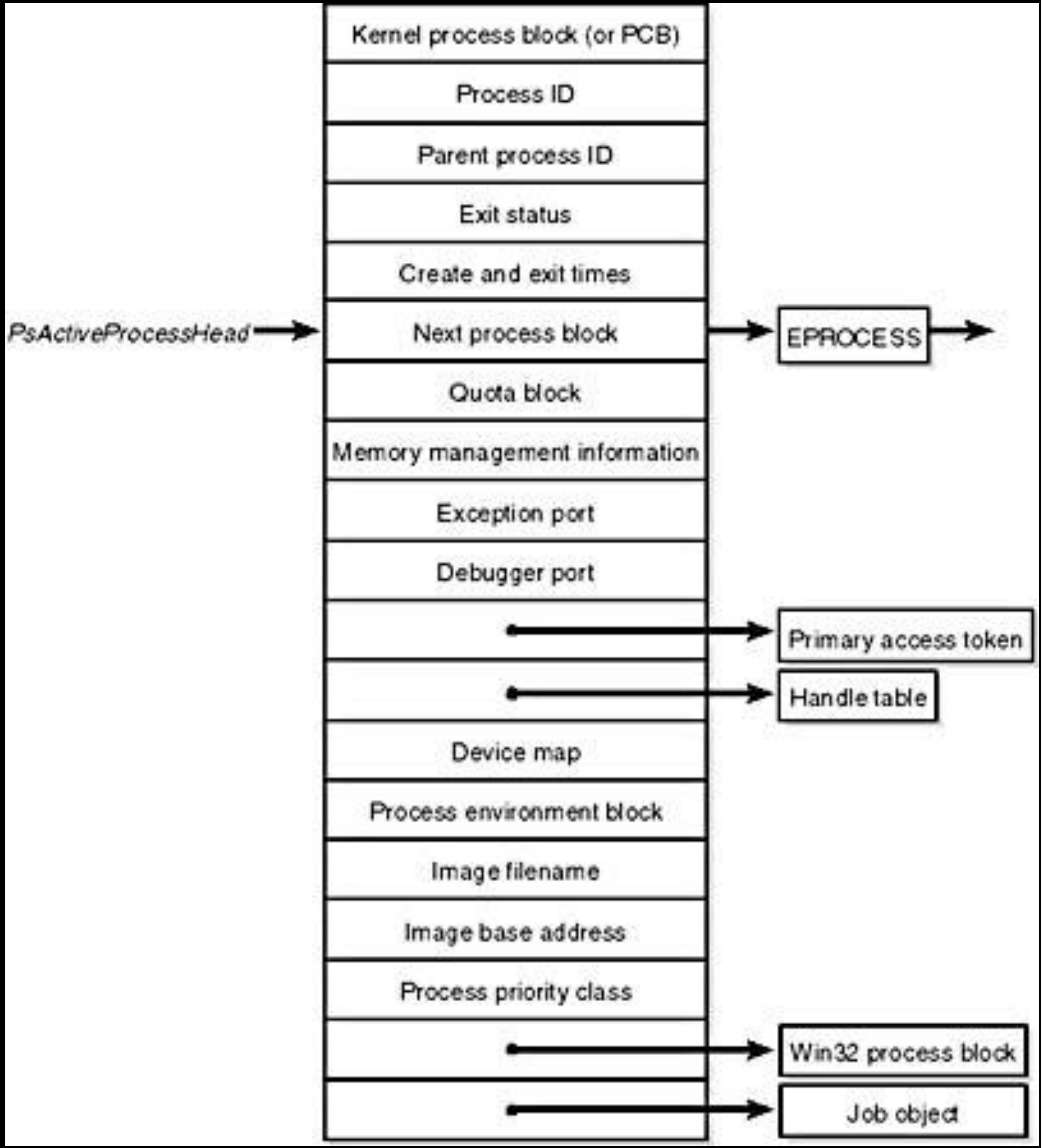
Process continued

Windows contains an overall structure of EProcess that is a wrapper for everything about a process.

Inside that is another struct called KProcess.

Linux

- basically operates in user space
- uses syscall.
- It also considers threads as lightweight processes

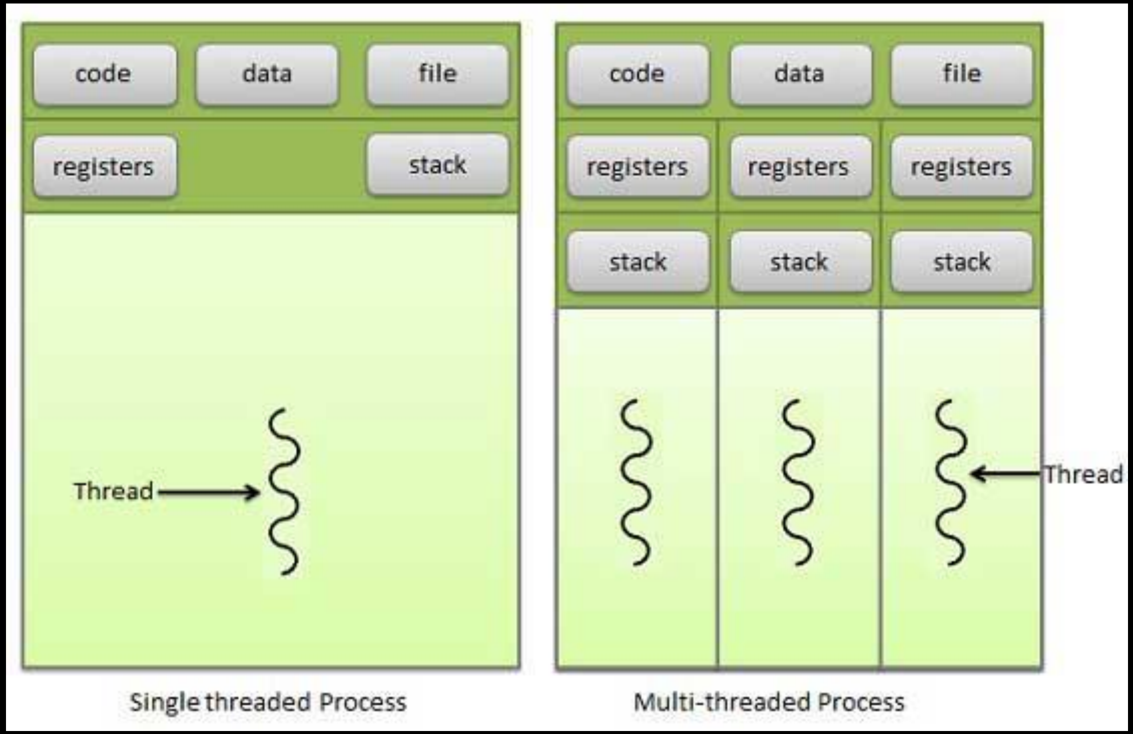


Thread

Each thread has stack memory allocated to it.

Threads contain information as follows:

- The I/O devices that will be used by the thread
- Whether the thread is synchronous or asynchronous
- Contains information about the two access levels
  - Kernel - protected access
  - User - general access



Kernel

Slower to create and manage

OS creates them

Specific to OS

Can be multithreaded

Descriptor Priviledge Level 0 - 2

```
typedef struct _KPROCESS {
DISPATCHER_HEADER Header;
LIST_ENTRY ProfileListHead, ReadyListHead, ThreadListHead, ProcessListEntry;
KGDTENTRY LdtDescriptor;
KIDTENTRY Int21Descriptor;
WORD IopmOffset;
UCHAR Iopl, Unused, State, ThreadSeed, PowerState, IdealNode, Visited;
ULONG ActiveProcessors, KernelTime, UserTime, ProcessLock, Affinity,
StackCount, DirectoryTableBase, Unused0;
SINGLE_LIST_ENTRY SwapListEntry;
PVOID VdmTrapcHandler;
union { ULONG AutoAlignment: 1; ULONG DisableBoost: 1; ULONG
DisableQuantum: 1; ULONG ReservedFlags: 29; LONG ProcessFlags; };
CHAR BasePriority, QuantumReset;
union { KEXECUTE_OPTIONS Flags; UCHAR ExecuteOptions; };
UINT64 CycleTime;
} KPROCESS, *PKPROCESS;
```

User

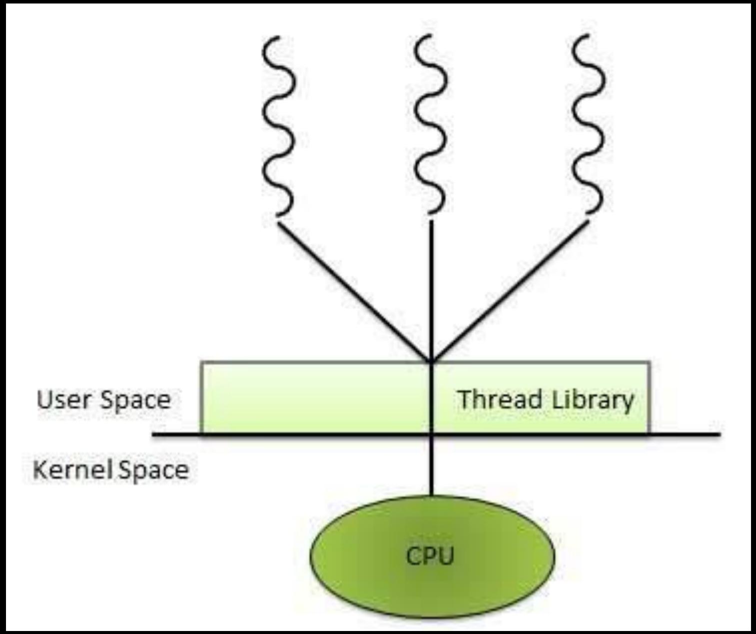
Run on any operating system

Operates only on one processor

Run independent of the Kernel

Uses cooperative multitasking, which means one thread blocks the others

Descriptor Priviledge Level 3



Relationship

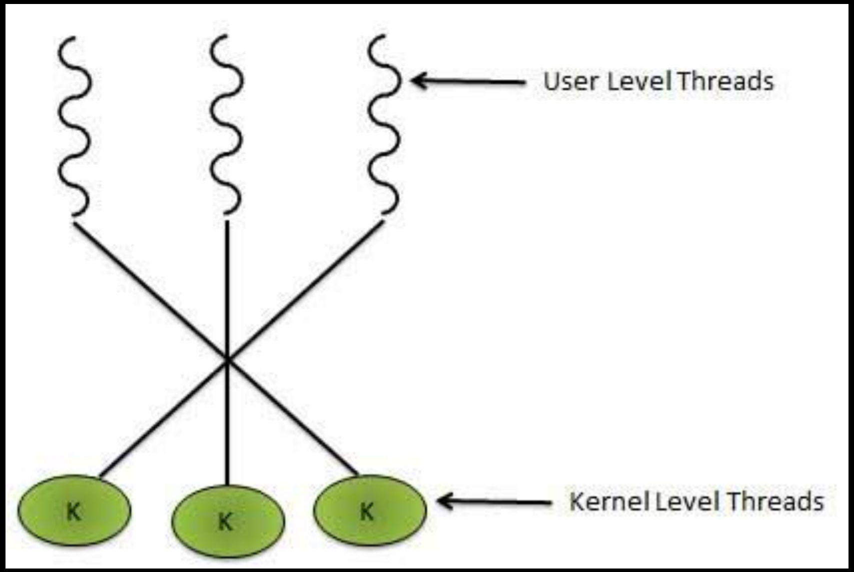
Between kernel and user

- Many to Many
- Many to One
- One to One

Many to Many

Multiple user level threads multiples with the same or smaller amount of kernel level threads.

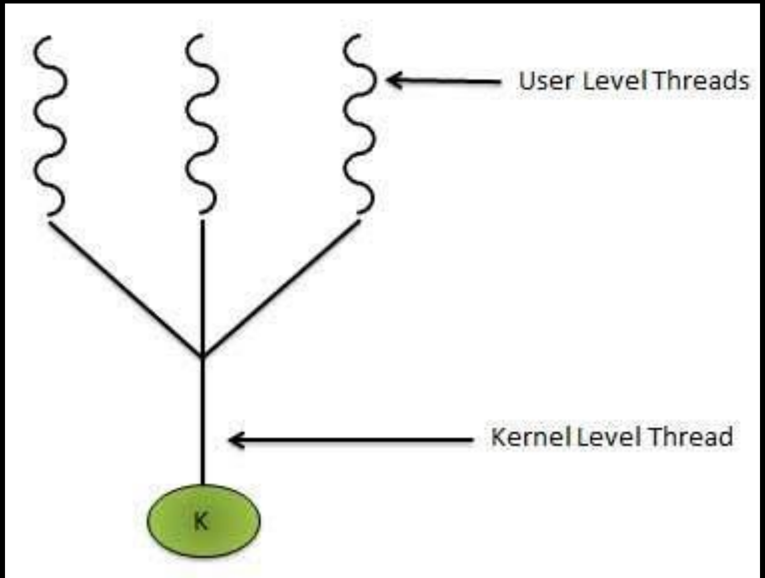
The number of kernel level threads are managed by the OS.



Many to One

Multiple user threads access the only one kernel.

More secure and eliminate race conditions, yet may cause deadlocks

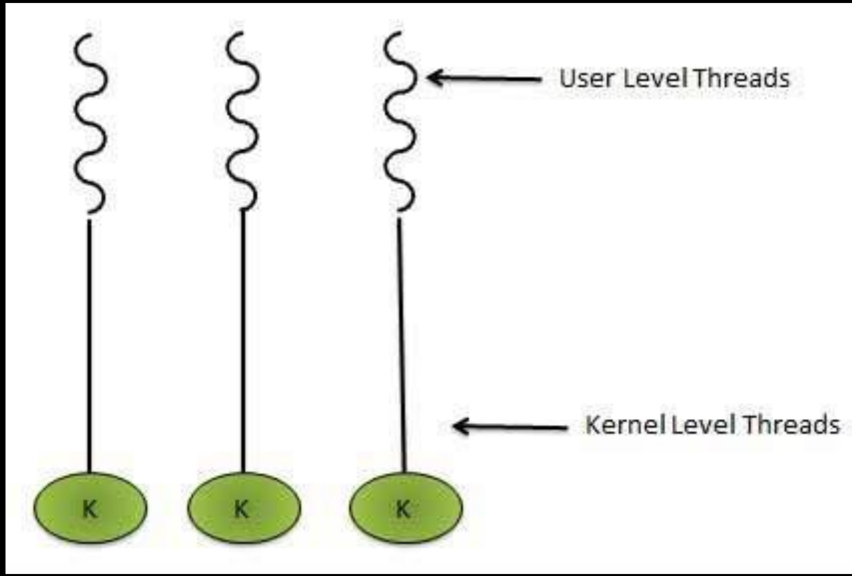


One to One

Better concurrency and support of parallelism.

Disadvantage is that two threads need to be created

Also for large tasks will only rely on one thread to do the work



Synchronization

Locks

- Semaphore
- Mutex
- Barrier
- Auto Locks
- Spin Lock

Semaphore

Uses a counting process to make sure that there is enough resources for each thread to execute.

Main purpose is to control I/O access in Operating Systems.

Portable and usually efficient

function V(semaphore S, integer I):

[S ← S + I]

function P(semaphore S, integer I):

repeat:

[**if** S ≥ I: S ← S − I **break**]

Producer-Consumer Application

Produce

P(emptyCount)

P(useQueue)

putItemIntoQueue(item)

V(useQueue)

V(fullCount)

Consume

P(fullCount)

P(useQueue)

item ← getItemFromQueue()

V(useQueue)

V(emptyCount)

Mutex

Binary Semaphore

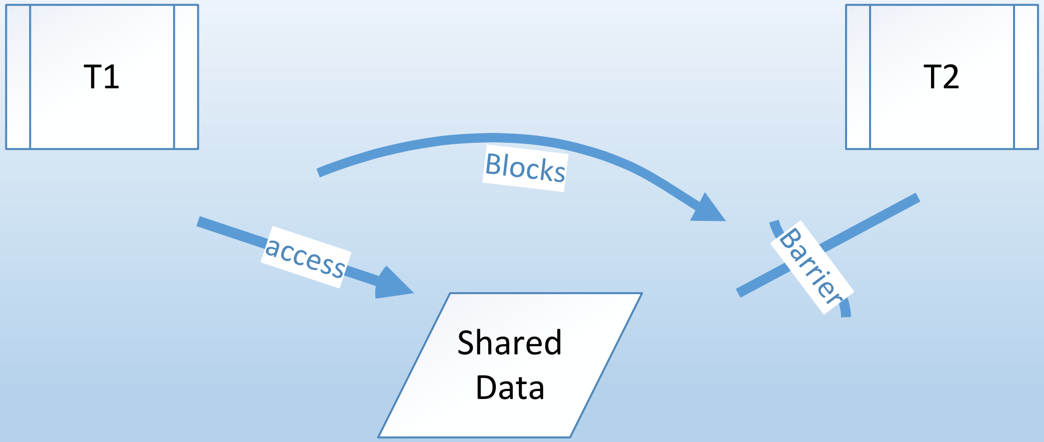
Mutual Exclusion

Only has two states

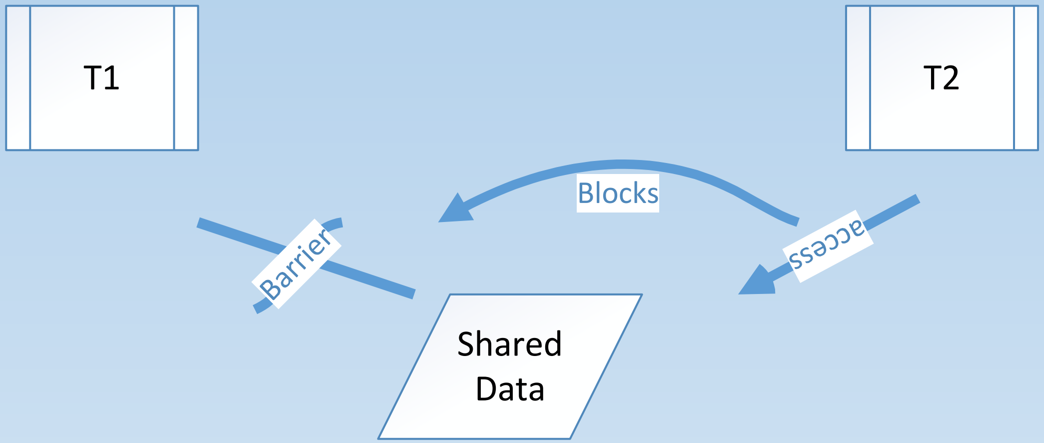
Only one thread will have access

Others will wait

```
semaphore mutex = 1.
while (1) {
        {0<=mutex<=1}
        P(mutex); -- entry protocol
        {mutex==0}
        CS
        V(mutex); -- exit protocol
        {0<=mutex<=1}
}
```



T1 completes and yields control



Barrier

- Optimizes the access to shared memory.
- Threads are required to wait until all threads have reached a certain point, the barrier.

```
semaphore here=0, go[1:2] = {0,0}
co
	while 1 {
		beforebarrier1;
		V(here); P(go[1]);
	}
	//
	while 1 {
		beforebarrier2;
		V(here); P(go[2]);
	}
	// -- coordinator
	while 1 {
		for [i=1,2] {
				P(here)
		};
		for [i=1,2] {
				V(go[i])
		}
	}
oc
```

Auto Locks

- Abstracted Mutex
- Locks once ownership comes into scope and unlocks when it goes out of scope

```
class AutoLock
{
public:
        AutoLock(Mutex *mutex)
        {
                if (mutex)
                {
                        m_mutex = mutex; m_mutex->lock();
                }
        }
        ~AutoLock()
        {
                if(m_mutex)
                {
                        m_mutex->unlock();
                }
        }
private:
        Mutex* m_mutex;
};
```

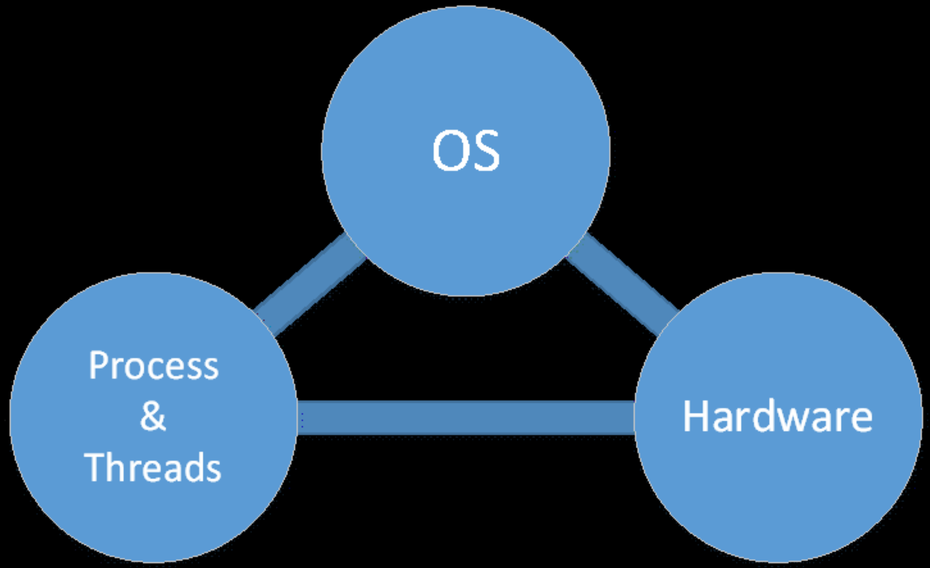
Spin Locks

- If the resource is taken by one thread, the second thread will be put in a loop to test when that resource is available.
- Require fewer resources to block a thread
- Better than putting the thread in suspended state

```
#include <pthread.h>
pthread_spinlock_t lock;
int pshared;
int ret;
/* initialize a spin lock */
 ret = pthread_spin_init(&lock,
pshared);
```

Proposed Solution

- Thread
- DynamicLock
- STL



Thread

- Additional Features from C++ Standard
  - volatile unsigned int cpu_affinity
  - unsigned long code_size;
  - volatile unsigned long stack_size;
  - unsigned int priority;
  - volatile int thread_state;
  - volatile double threshold;
  - volatile double _ticks, _duration;
  - Many more

DynamicLock

```
template< … Args volatile>
public class ILock
{
        …Args variables;  // example unsigned int cpu_cores
        struct _secAttr;
public:
        // do not want to copy or move for security reasons
        DynamicLock(const DynamicLock & obj) = delete;
        DynamicLock(DynamicLock && obj) = delete;
        DynamicLock & operator=(const DynamicLock& obj) = delete;
        DynamicLock & operator=(DynamicLock&& obj) = delete;

        void Lock();
        void Unlock() ;
};
```

STL - Proposed

```
Template<class  T, class  U, … Args>
public  _threadsync  : class T
{
        volatile  _thread*  next;
        volatile  double  _weight;
        DynamicLock<…  Args> *lock;
        T<std::thread*>  _thread_container;
        template  <… > U<std::futures<…>>  _async_container;

        bool update_thread(_thread  *_updated);

        _utry();
        _kcatch();

        execute((void  *)fptr);
        execute(std::function<void>());

        void* fptr;
        std::function<void>  func;
public:

        bool add_thread(_thread&  new_thread);
        void remove_thread(_thread&  removed);
        template<…  >bool add_handler((void  *)fptr<… >(… args));
        void define_execution((void  *) _fp);
        void define_execution(std::function<void>  _fn);
}
```

execute(???)

```
void execute((void *) _fp)
{
        _utry{
                _fp();
        }
        _kcatch
        {
                std::future<…> _clean();
                std::future<…> _realloc();
                        .
                        .
                        .
                std::future<…> _writelog();
        }

}
```

Examples

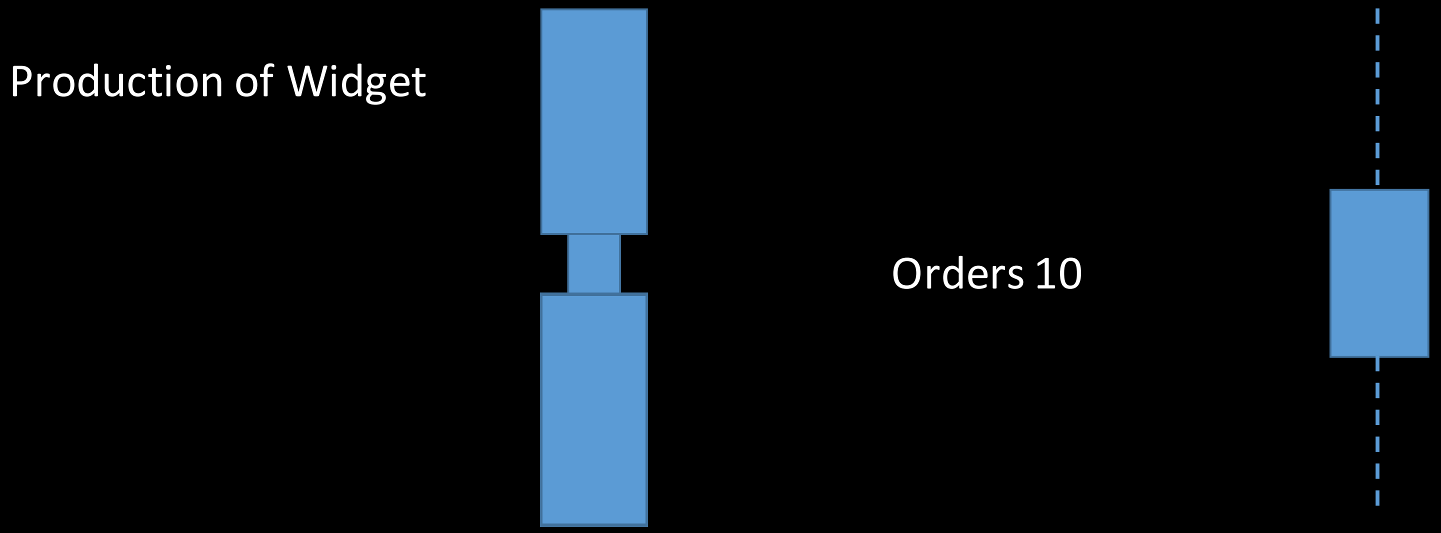
Producer-Consumer

Inventory

15

Producer

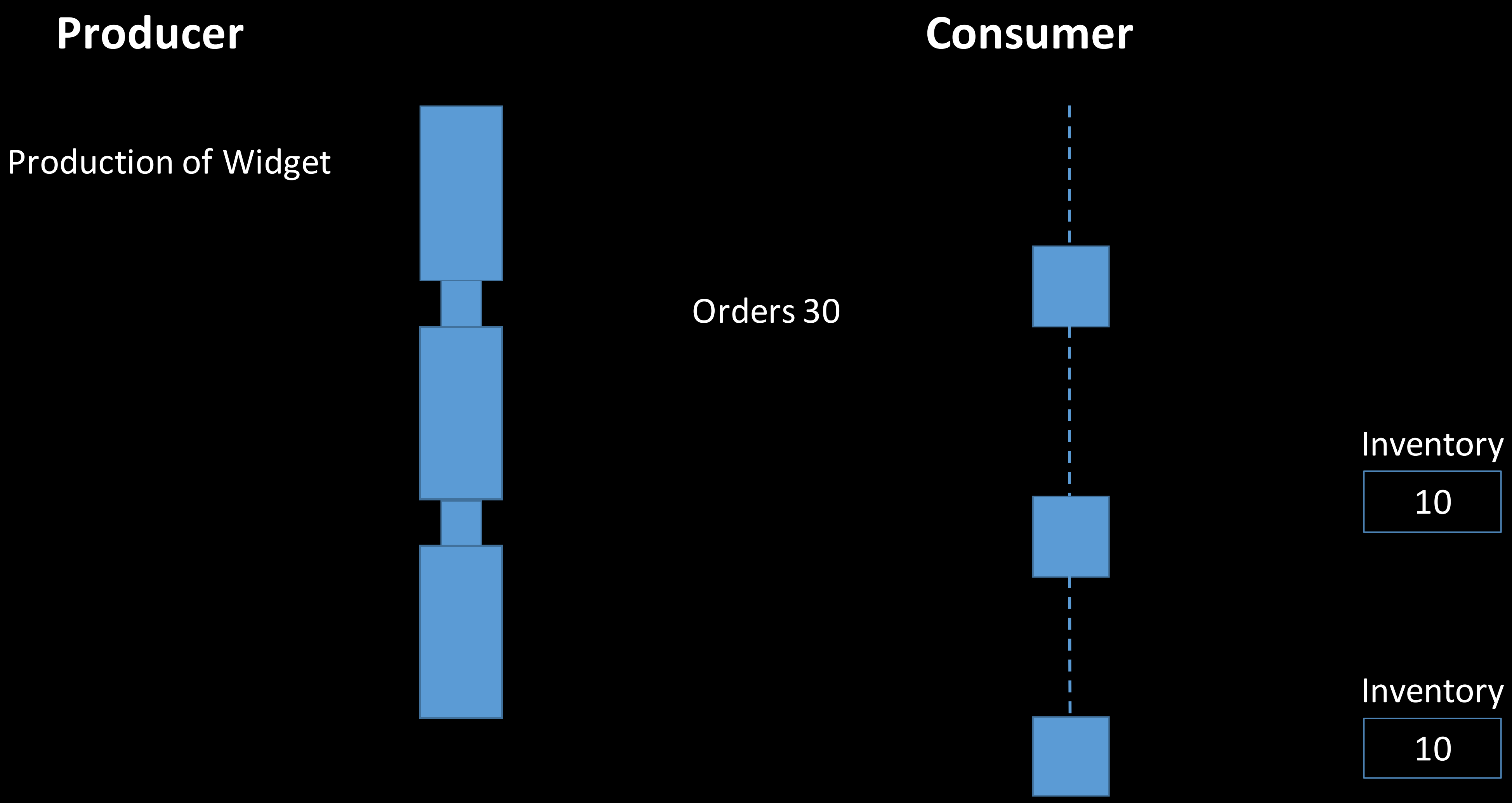
Consumer



Producer-Consumer

Inventory

10

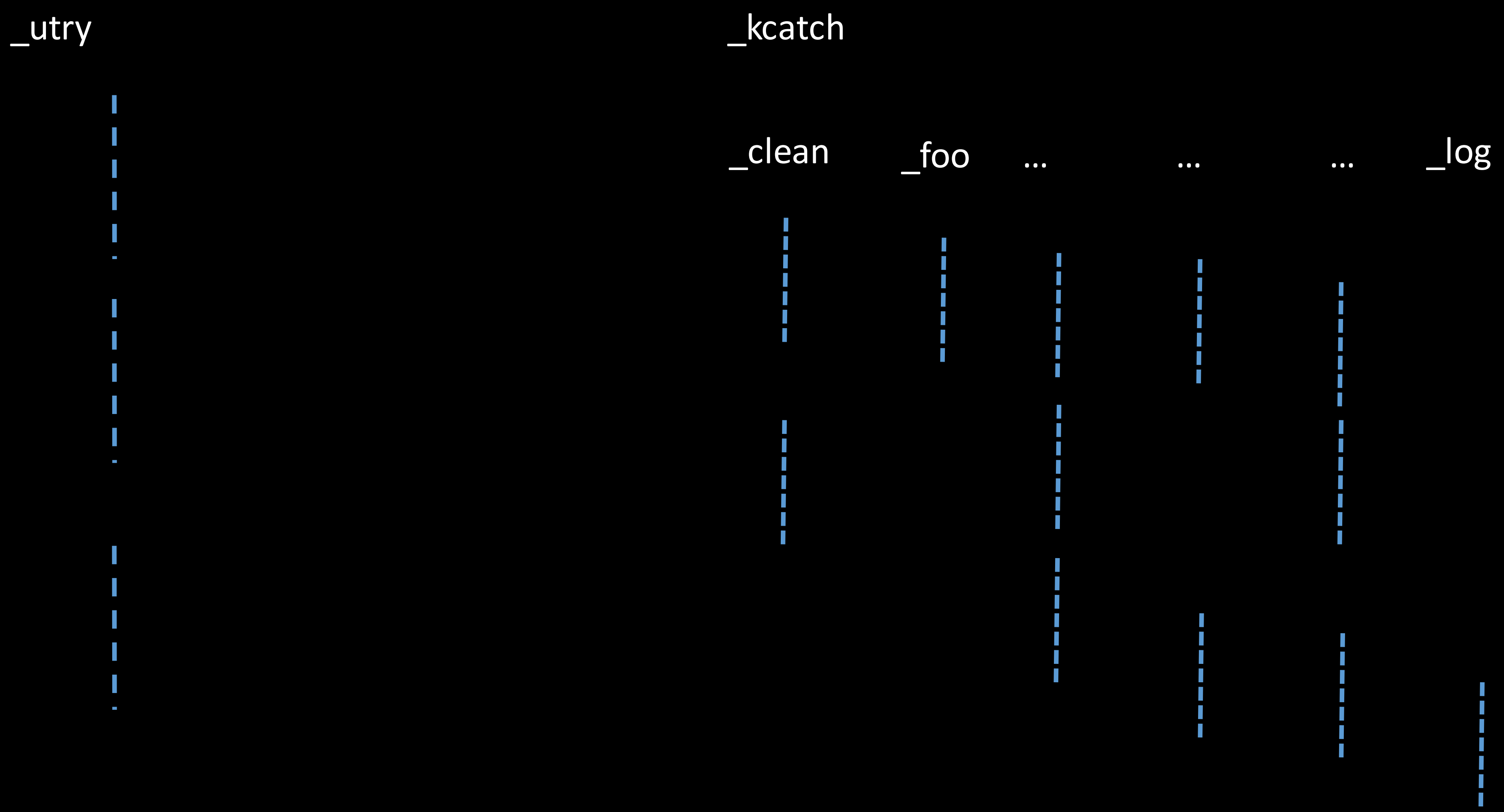


Producer-Consumer

Inventory

10

Game Play



Other Scenarios

- Another production line is put in place
  - Spin Lock
  - Heap memory with a const pointer, volatile data
- Two customers order, one customer needs the order filled in a hurry and a limited supply
  - Latch for second customer
  - Mutex for first customer

Conversation on the SG14 forum

The committee has no direct influence over OS or hardware vendors. If there is something provided by multiple vendors, IMO it is fair game to propose an abstraction that covers it. If it is performance-related (especially in a non-obvious way), an implementation across multiple platforms and timing measurements showing a clear benefit in typical use cases for it go a long way towards selling your proposals.

Yes, it does but not in a direct way. Vendors have been bending over to adapt to the 'standard way' to program C/C++. See Intel itself, CUDA, Microblaze/NIOS, and the whole C-to-HDL ecosystem (https://en.wikipedia.org/wiki/C_to_HDL) just to mention a few from the top of my head. The committee has absurd attention and quasi-religious following among software professionals to have indeed a huge (unintended) influence. Want it or not, you are part of a strong feedback circle that affects how hardware technologies are implemented (or not) in the future. As an example, if we had zero-copy semantics adopted when it came out long ago, perhaps I would not have to be doing the LD_PRELOAD hack today.

HFT has been about collapsing layers of software and hardware. Application/presentation/session/transport/network and sometimes eve data layers today reside in a single binary blob through a whole hack-a-mole pipeline. And all this happened while the C++ comittee was focused developing auto, lambda and variadics.

I think SG14 is a good opening to close more this circle and engage with the industry, pretty much like the MPI standard embraced the HPC industry. But I can see how that can be a turn off for many.

References

- http://www.futurechips.org/chip-design-for-all/cpu-vs-gpgpu.html
- http://www.embedded.com/design/prototyping-and-development/4231326/Taking-advantage-of-the-Cortex-M3-s-pre-emptive-context-switches
- **Building Parallel, Embedded, and Real-Time Applications with Ada** John W. McCormick, Frank Singhoff, Jérôme Hugues
- http://www.tutorialspoint.com/operating_system/os_process_scheduling.htm
- https://www.cs.rutgers.edu/~pxk/416/notes/07-scheduling.html
- https://computing.llnl.gov/tutorials/pthreads/
- http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4399.html
- http://www.boost.org/doc/libs/1_59_0/doc/html/lockfree.html
- https://software.intel.com/en-us/articles/implementing-scalable-atomic-locks-for-multi-core-intel-em64t-and-ia32-architectures
- https://software.intel.com/en-us/articles/introduction-to-x64-assembly
- http://www.keil.com/support/man/docs/armasm/armasm_dom1359731128950.htm
- *www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/**n4582**.pdf*
- *https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/5_CPU_Scheduling.html*
- *http://akosma.com/2010/10/11/how-knowing-c-and-c-can-help-you-write-better-iphone-apps-part-1/*
- *http://www.plantation-productions.com/Webster/www.artofasm.com/Linux/HTML/MemoryAccessandOrg.html*
- *http://www.tldp.org/LDP/tlk/ds/ds.html*
- *http://www.tutorialspoint.com/operating_system/os_multi_threading.htm*

- http://deploytonenyures.blogspot.com/2013/10/windows-vs-linux-processes-and-threads.html
- http://www.linfo.org/kernel_space.html
- http://www.codeproject.com/Articles/421976/How-to-create-a-Simple-Lock-Framework-for-Cplusplu
- http://en.cppreference.com/w/cpp/language/parameter_pack
- http://en.cppreference.com/w/cpp/thread/future
- http://en.cppreference.com/w/cpp/thread/async
- https://en.wikipedia.org/wiki/Asymmetric_multiprocessing
- http://searchdatacenter.techtarget.com/definition/SMP
- http://www.gulamshakir.com/2012/10/03/C%2B%2B-Multithreading-Tips.html
- https://docs.oracle.com/cd/E26502_01/html/E35303/ggecq.html
- http://www.thegeekstuff.com/2012/03/linux-threads-intro/
- http://fpgacenter.com/fpga/fpga_arch.php
- https://www.safaribooksonline.com/library/view/designing-embedded-hardware/0596007558/ch01.html
- http://stackoverflow.com/questions/6155951/whats-the-difference-between-deadlock-and-livelock
- http://stackoverflow.com/questions/3111403/what-is-the-difference-between-a-lock-and-a-latch-in-the-context-of-concurrent-a
- http://www.dba-oracle.com/t_lru_latches.htm
- http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n3998.html
- https://en.wikipedia.org/wiki/Semaphore_(programming)
- http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3666.html
- http://superuser.com/questions/455316/what-is-a-user-thread-and-a-kernel-thread