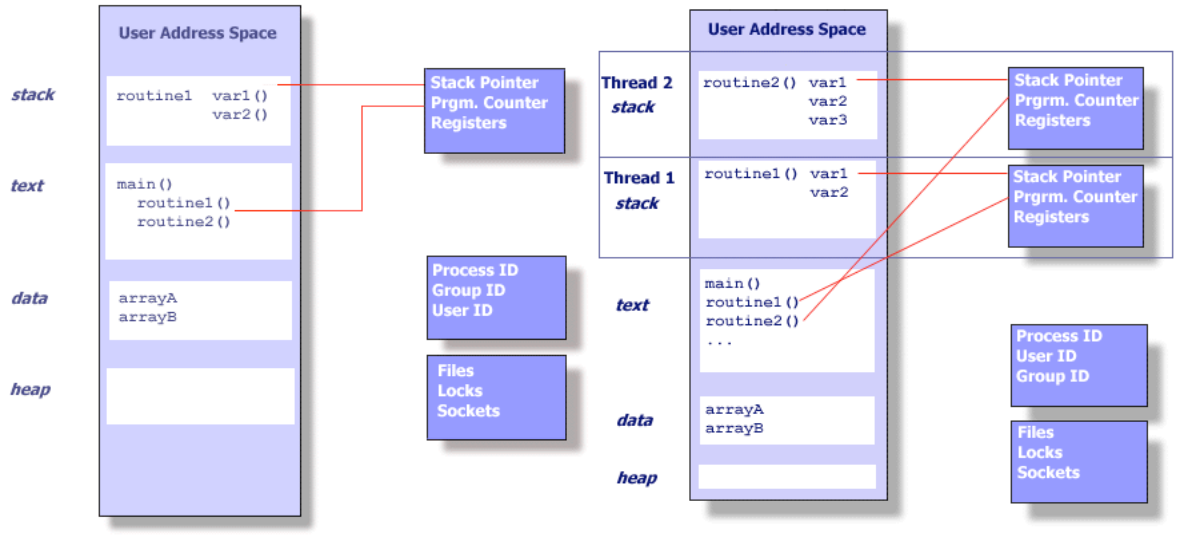
Lecture 28: Board Notes: Threads and GPUs

Part A:

Threads vs. Processes in Pictures:



- **Part A:** Thus, in a UNIX environment a thread:
  - Exists within a process, and uses process resources
  - Has its own independent control flow (PC)
  - Duplicates only resources it needs to be executed by processor datapath
  - Shares process resources with other threads
    - Thus, if you close a file with one thread, all threads see file closed
  - Dies if parent process dies

##

Part B:

(Some of the) advantages of threads:

- Can be much faster to schedule / run on HW
  - Avoid overhead of a context switch

- Relatively speaking, process context switch requires considerable work
  - 1000s of ns! Can execute instruction in just a few ns or less…
  - Side note: Why?
    - Recall VA translation:
      - TLB → Cache → deliver data (in best case)
        - Each process has its own Page Table
        - TLB is fast cache for Page Table
        - TLB has at least 64 entries
        - TLB misses require main memory accesses…

- Other benefits:
  - Responsiveness:
    - If one thread is blocked or writing, others can work
  - Resource Sharing:
    - Application has several threads of activity in one 1 address space
  - Economy:
    - In Solaris 2, crating a process 30X slower than thread, switching 5x slower
  - Utilization of multiprocessors:
    - Each thread can run in parallel on a different processor core

##

Part C:

Hardware Models:

Generically, hardware multi-threading allows multiple threads to share functional units of single processor in an overlapping fashion

- Each processor has to duplicate independent state of each thread
  - i.e. register file and PC
- Memory can be shared through virtual memory mechanisms which already could support multi-programming

Picture of hypothetical threaded datapath:

Ideally, HW should support relatively quick change to new thread very quickly

- i.e. as opposed to process context switch

Part D:

3 models of multi-threading to discuss:

- **Fine grained multi-threading:**
  - Might switch between threads EVERY instruction
  - Interleaving done in round robin fashion
    - Threads that are stalled could just be skipped
  - Practically, MUST be able to switch threads each CC
    - Therefore, would probably have separate register files – and would not do a register file content swap

- Advantages:
  - Can high throughput losses from stalls…
  - i.e. could eliminate stall of lw followed by ALU in MIPS 5 stage pipe
- Disadvantage
  - Slows down execution of individual (higher priority?) threads
  - i.e. each thread gets its turn

Sketch of datapath with multiple PC registers, register files…

Hypothetical threads and pipeline:

T1: a. lw $5, 0($6) T2: c. sub $10, $11, $12
b. add $7, $5, $9

Pipeline:

		1	2	3	4	5	6	7	8
a.	lw …	F	D	E	M	W			
c.	sub…		F	D	E	M	W		
b.	add…			F	D	E	M	W	

- **Course grained multi-threading:**
  - Only switches threads on "expensive" stalls – i.e. L2 cache miss
  - Advantages:
    - Can simplify thread swap overhead; could take longer
    - Individual threads not slowed as much
  - Disadvantages:
    - Harder to overcome throughput losses from shorter stalls
      - i.e. on a stall, pipeline must be emptied or frozen and then refilled
      - *Thus, pipeline drain/fill time should be small compared to stall time*

- **Simultaneous multi-threading:**
  - A variation on HW multi-threading that uses the resources of superscalar machines
  - Exploit both instruction level parallelism (i.e. pipelining) and thread-level parallelism
  - Idea:
    - Multi-issue superscalar processors often have more functional unit parallelism available than 1 thread can use effectively
    - Moreover, with register renaming (mapping tables), dynamic scheduling, etc. multiple instructions can be issued from multiple threads without worrying about dependencies between them.

Remember register-renaming idea – i.e. have 1 large register file with remaps

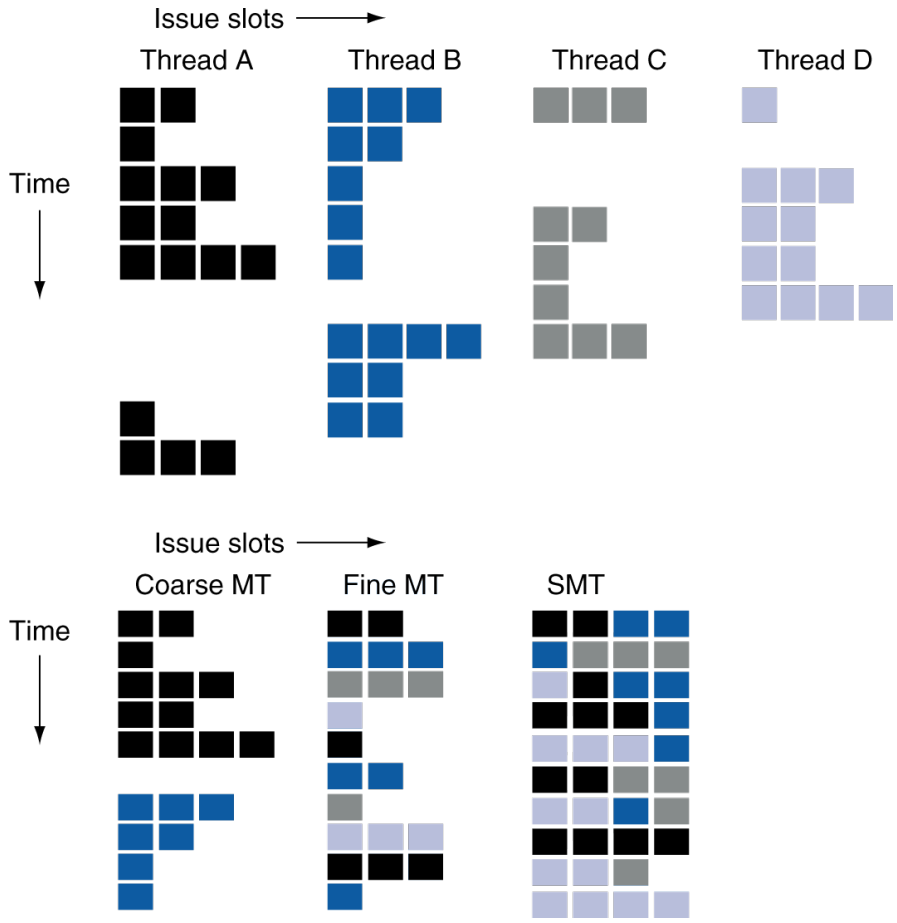
Assume: Map Table:

	R1	R2	R3
	q7	q8	q9
Add R1, R2, R3 →	q10	q8	q9

Could say that q0 → q31 devoted to Thread 1, q31 → q62 devoted to Thread 2, etc.

Could also share data between threads this way.

Example 1: Fine grained vs. Coarse grained vs. SMT:



##

Part E:

Example 2:

Thread 1:		**Thread 2:**		**Thread 3:**	
a.	Add R1, R2, **R3**	d.	Sub R7, R8, R9	g.	Load **R3**, 0(R2)
b.	Div R8, R10, R12	e.	XOR R21, R22, R23	h.	Add R11, R3, R1
c.	Add R17, R8, R19	f.	AND R24, R24, R7	i.	Add R12, R3, R2

Consider the following:

- R3 referenced in T1 and R3 referenced in T3 could be 2 different physical registers
  - Could also have 1 common register file and do re-mappings
- With fine grain model, each thread could get a turn in some round-robin fashion
  - Having separate register files would significantly reduce context switches
  - Instruction **a** could immediately be followed by instruction **d** in the pipeline
  - Could skip T3 if waiting for page fault because of instruction **g**
  - **Did example previously.**
- With coarse grain model, may do context switch if:
  - Instruction **b** has high latency and need to wait on RAW hazard
  - Instruction **g** has L2 cache miss and need to wait to process
- With SMT, Instructions **a, b, d,** and **g** may run simultaneously
  - Realistic modern superscalar datapath might have a lw/sw unit, a few adders, 1 multipler, 1 divider, etc.
  - *Draw picture of parallel execution paths.*

Part F:

Example 2:

Consider the following three CPU organizations:

- CPU SS:
  - A 2-core superscalar microprocessor that provides out-of-order issue capabilities on two functional units. Only a single thread can run on each core at a time
- CPU MT:
  - A fine-grained multi-threaded processor that allows instructions from two threads to be run concurrently (i.e. there are two functional units), though only instructions from a single thread can be issued on any cycle.
- CPU SMT:
  - An SMT processor that allows instructions from two threads to be run concurrently (i.e. there are two functional units), and instructions from either or both threads can be run on any cycle.

Assume we have two threads X and Y to run on these CPUs that include the following operations:

Thread X	**Thread Y**
A1 – takes 2 cycles to execute	B1 – no dependencies
A2 – depends on the result of A1	B2 – conflicts for a functional unit with B1
A3 – conflicts for a functional unit with A2	B3 – no dependencies
A4 – depends on the result of A2	B4 – depends on the result of B2

Assume all instructions take a single cycle to execute unless noted otherwise OR they encounter a hazard.

Question 1:
Assume that you have one SS CPU. How many cycles will it take to execute these 2 threads? How many issue slots are wasted due to hazards?

Core 1	**Core 2**
A1, A3	B1, B3
A1	B2
A2	B4
A4	

9 of 16 slots used, 56% usage rate

Question 2:
Now assume that you have 1 MT CPU. How many cycles will it take to execute these 2 threads? How many issue slots are wasted due to hazards? Assume each thread gets 2 CCs

Functional Unit 1	**Functional Unit 2**
A1	A3
A1	
B1	B3
B2	
A2	
A4	
B4	

9 of 14 slots used for a 64% usage rate; could also argue 9 of 16 slots used for 54% usage rate.

Question 3:
Now assume that you have 1 MT CPU. How many cycles will it take to execute these 2 threads? How many issue slots are wasted due to hazards?

Functional Unit 1	**Functional Unit 2**
A1	B1
A1	B2
A2	B3
A3	B4
A4	

##

Part G:

Writing threaded programs for supporting HW:

- For UNIX systems, a standardized, C-language threads programming interface has been specified by the IEEE – POSIX threads (or Pthreads)
- For a program to take advantage of Pthreads…
    - Should be capable of being organized into discrete, independent tasks which can execute concurrently
- More detailed discussion in Lecture 26…
- …but generally, programs that have the following characteristics are well-suited for Pthreads:
    - Work that can be executed OR data that can be operated on multiple tasks at the same time
    - Have sections that will block and experience long I/O waits
        - i.e. while 1 thread is waiting for I/O system call to complete, CPU intensive work can be performed by other threads
    - Use many CPU cycles in some places, but not others
    - Must respond to asynchronous events
        - i.e. a web server can transfer data from previous requests and manage arrival of new requests
    - Some work is more important than others (priority interrupts)
- SUNSparc T2 (Niagra T2) – for servers, circa October 2007
    - Simple cores, fine-grained model
    - Need to know so that if you have a high-priority thread, you can schedule accordingly