Lec 26: Parallel Processing

Kavita Bala
CS 3410, Fall 2008
Computer Science
Cornell University

Announcements

- Pizza party
  - Tuesday Dec 2, 6:30-9:00
  - Location: TBA
- Final project (parallel ray tracer) out next week
  - Demos: Dec 16 (Tuesday)
- Prelim 2: Dec 4 Thursday
  - Hollister 110, 7:30-9:00/9:30

Amdahl’s Law

- Task: serial part, parallel part
- As number of processors increases,
  - time to execute parallel part goes to zero
  - time to execute serial part remains the same
- *Serial part eventually dominates*
- Must parallelize ALL parts of task

$$\text{Speedup}(E) = \frac{\text{Execution Time without } E}{\text{Execution Time with } E}$$

Amdahl’s Law

- Consider an improvement E
- F of the execution time is affected
- S is the speedup

$$\text{Execution time (with } E) = ((1-F) + F/S) \cdot \text{Execution time (without } E)$$

$$\text{Speedup (with } E) = \frac{1}{(1-F) + F/S}$$

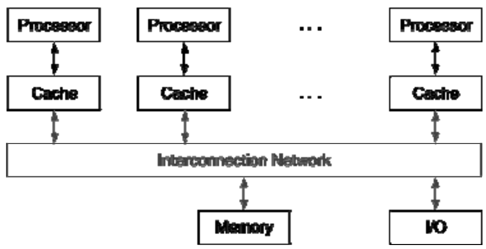
Amdahl's Law

- Sequential part can limit speedup
- Example: 100 processors, 90× speedup?
  - $T_{new} = T_{parallelizable}/100 + T_{sequential}$
  - $$\text{Speedup} = \frac{1}{(1 - F_{parallelizable}) + F_{parallelizable}/100} = 90$$
  - Solving: $F_{parallelizable} = 0.999$
- Need sequential part to be 0.1% of original time

© Kavita Bala, Computer Science, Cornell University

Shared Memory

- SMP: shared memory multiprocessor
  - Hardware provides single physical address space for all processors
  - Synchronize shared variables using locks



Cache Coherence Problem

- Suppose two CPU cores share a physical address space
  - Write-through caches

Time step	Event	CPU A's cache	CPU B's cache	Memory
0				0
1	CPU A reads X	0		0
2	CPU B reads X	0	0	0
3	CPU A writes 1 to X	1	0	1

Coherence Defined

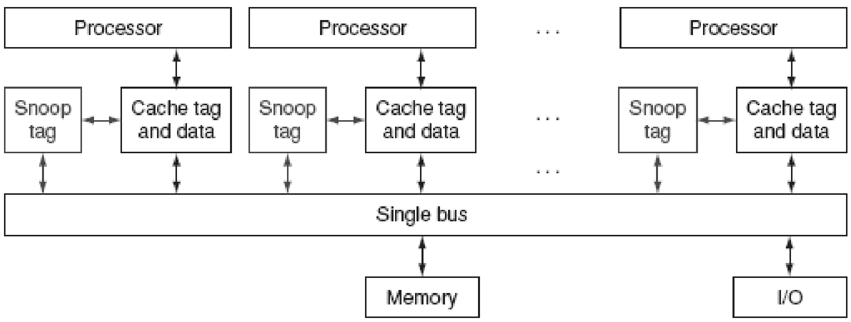
- Informally: Reads return most recently written value
- Formally:
  - P writes X; P reads X (no intervening writes) $\Rightarrow$ read returns written value
  - P_1 writes X; P_2 reads X (sufficiently later) $\Rightarrow$ read returns written value
    - CPU B reading X after step 3 in example
  - P_1 writes X, P_2 writes X $\Rightarrow$ all processors see writes in the same order
    - End up with the same final value for X
    - Sequential consistency

Cache Coherence Protocols

- Operations performed by caches in multiprocessors to ensure coherence
  - Migration of data to local caches
    - Reduces bandwidth for shared memory
  - Replication of read-shared data
    - Reduces contention for access
- Snooping protocols
  - Each cache monitors bus reads/writes

Snooping Caches

- Read: respond if you have data
- Write: invalidate or update your data



Invalidating Snooping Protocols

- Cache gets exclusive access to a block when it is to be written
  - Broadcasts an invalidate message on the bus
  - Subsequent read in another cache misses
    - Owning cache supplies updated value

CPU activity	Bus activity	CPU A's cache	CPU B's cache	Memory
				0
CPU A reads X	Cache miss for X	0		0
CPU B reads X	Cache miss for X	0	0	0
CPU A writes 1 to X	Invalidate for X	1		0
CPU B read X	Cache miss for X	1	1	1

Writing

- Write-back policies for bandwidth
- Write-invalidate coherence policy
  - First invalidate all other copies of data
  - Then write it in cache line
  - Anybody else can read it
- Permits one writer, multiple readers

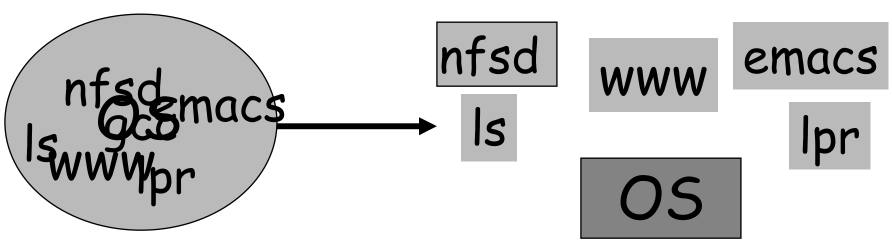
- In reality: many coherence protocols
  - Snooping doesn't scale
  - Directory-based protocols
    - Caches and memory record sharing status of blocks in a directory

© Kavita Bala, Computer Science, Cornell University

Parallel Programming and Synchronization

Processes

- Hundreds of things going on in the system: how to manage?



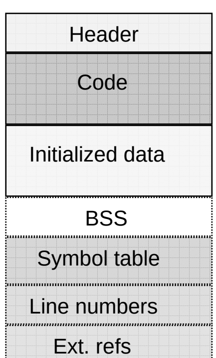
- How to make things simple?
  - Decompose computation into separate *processes*
- How to make things reliable?
  - Isolate processes from each other to protect from each others' faults
- How to speed up?
  - Overlap I/O bursts of one process with CPU bursts of another

What is a process?

- A program being executed
  - Sequential, one instruction at a time
- Process is an OS abstraction
  - a thread of execution running in a restricted virtual environment – a virtual CPU and virtual memory environment, interfacing with the OS via system calls
  - The unit of execution
  - The unit of scheduling
  - Thread of execution + address space

The same as “job” or “task” or “sequential process”. Closely related to “thread”

Process != Program

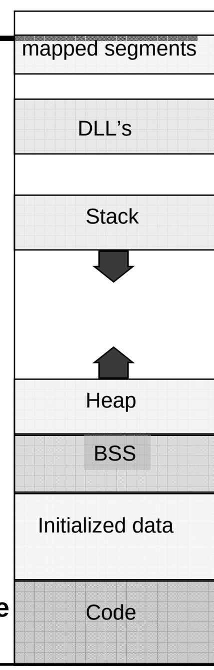


Executable

Program is passive
• Code + static data

Process is running program
• stack, registers, heap, pc

Example:
We both run IE on one machine
- same program
- separate processes
- same virtual address space
- different physical memory



Process address space

Context Switch

- Context Switch
  - Process of switching CPU from one process to another
- State of a running process must be saved and restored:
  - Program Counter, Stack Pointer, General Purpose Registers
- Suspending a process: OS saves state
  - Saves register values
- To execute another process, the OS restores state
  - Loads register values

Details of Context Switching

- Context switching code is architecture-dependent
  - Depends on registers
- Very tricky to implement
  - OS must save state without changing state
  - Must run without changing any user program registers
    - CISC: single instruction saves all state
    - RISC: reserve registers for kernel
- Overheads: CPU is idle during a context switch
  - Explicit:
    - direct cost of loading/storing registers to/from main memory
  - Implicit:
    - Opportunity cost of flushing useful caches (cache, TLB, etc.)
    - Waiting for pipeline to drain in pipelined processors

© Kavita Bala, Computer Science, Cornell University

How to create a process?

- Double click on a icon?
- After boot OS starts the first process
  - E.g., init
- The first process creates other processes:
  - the creator is called the parent process
  - the created is called the child process
  - the parent/child relationships creates a process tree

Processes Under UNIX

- New *child* process is created by the fork() system call:

  `int fork()`

  - creates a new address space
  - copies the parent's address space into the child's
    - uses copy-on-write to avoid copying memory that is only read
  - starts a new thread of control in the child's address space
  - parent and child are *almost* identical
    - in parent, fork() returns a non-zero integer
    - in child, fork() returns a zero.
    - difference allows parent and child to distinguish themselves
  - int fork() returns TWICE!

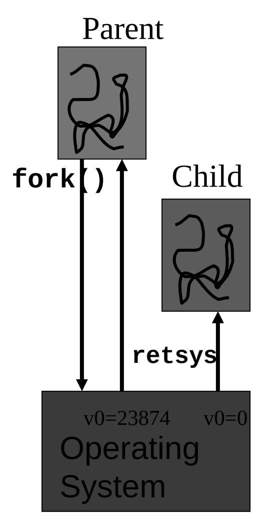
Example

```
main(int argc, char **argv)
{
   char *myName = argv[1];
   int cpid = fork();
   if (cpid == 0) {
      printf(“The child of %s is %d\n”, myName, getpid());
      exit(0);
   } else {
      printf(“My child is %d\n”, cpid);
      exit(0);
   }
}
```

What does this program print?

Bizarre But Real

```
lace:tmp<15> cc a.c
lace:tmp<16> ./a.out foobar
The child of foobar is 23874
My child is 23874
```



Cooperating Processes

- Processes can be independent or can work cooperatively
- Cooperating processes can be used for:
  - speedup by spreading computation over multiple processors/cores
  - speedup and improving interactivity: one process can work while others are stopped waiting for I/O.
  - better structuring of an application into separate concerns
    - E.g., a pipeline of processes processing data
- But: cooperating processes need ways to
  - Communicate information
  - Coordinate (synchronize) activities

Shared memory

- By default processes have disjoint physical memory -- complete isolation prevents communication
- Processes can set up a segment of memory as *shared* with other process(es)
  - Typically part of the memory of the process creating the shared memory. Other processes attach this to their memory space.
- Allows high-bandwidth communication between processes by just writing into memory

© Kavita Bala, Computer Science, Cornell University

Example

```
#include <stdio.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main(int argc, char **argv) {
  char* shared_memory;
  const int size = 4096;
  int segment_id = shmget(IPC_PRIVATE, size, S_IRUSR | S_IWUSR);
  int cpid = fork();
  if (cpid == 0) {
    shared_memory = (char*) shmat(segment_id, NULL, 0);
    sprintf(shared_memory, "Hi from process %d",getpid());
  } else {
    shared_memory = (char*) shmat(segment_id, NULL, 0);
    wait(NULL);
    printf("Process %d read: %s\n", getpid(), shared_memory);
    shmdt(shared_memory);
    shmctl(segment_id, IPC_RMID, NULL);
  }
}
```

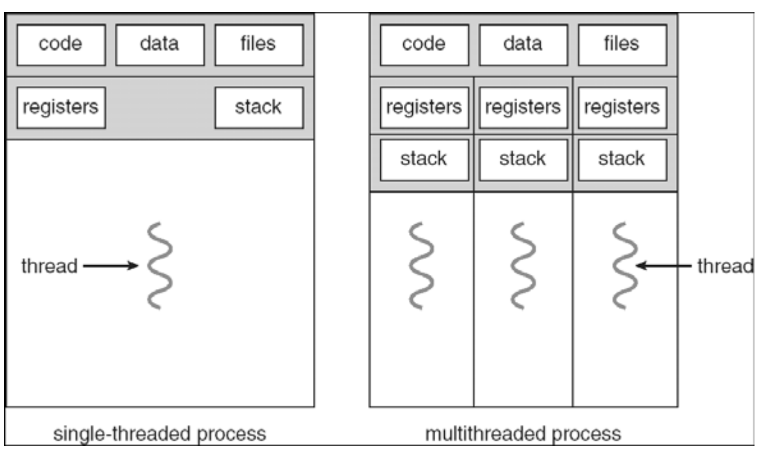

Processes are heavyweight

- Parallel programming with processes:
  - They share almost everything
  - They all share the same code and any data in shared memory (process isolation is not useful)
  - They all share the same privileges
- What don’t they share?
  - Each has its own PC, registers, and stack
- Idea: why don’t we separate the idea of process (address space, accounting, etc.) from that of the minimal “thread of control” (PC, SP, registers)?

Threads vs. processes

- Most operating systems therefore support two entities:
  - the process,
    - which defines the address space and general process attributes
  - the thread,
    - which defines a sequential execution stream within a process
- A thread is bound to a single process.
  - For each process, however, there may be many threads.
- Threads are the unit of scheduling
- Processes are *containers* in which threads execute

Multithreaded Processes



Threads

```
#include <pthread.h>
int hits = 0;

void *PrintHello(void *threadid) {
  int tid; tid = (int)threadid;
  printf("Hello World! It's me, thread #%d! hits %d\n",
   tid, ++hits);
  pthread_exit(NULL);
}
int main (int argc, char *argv[]) {
  pthread_t threads[5];
  int t;
  for(t=0; t<NUM_THREADS; t++){
    printf("In main: creating thread %d\n", t);
    pthread_create(&threads[t],NULL,PrintHello,(void *)t);
   }
   pthread_exit(NULL);
}
```


- If processes….

- If threads….

Programming with threads

- Need it to exploit multiple processing units
  …to provide interactive applications
  …to write servers that handle many clients
- Problem: hard even for experienced programmers
  - Behavior can depend on subtle timing differences
  - Bugs may be impossible to reproduce
- Needed: synchronization of threads

Goals

- Concurrency poses challenges for:
- Correctness
  - Threads accessing shared memory should not interfere with each other
- Liveness
  - Threads should not get stuck, should make forward progress
- Efficiency
  - Program should make good use of available computing resources (e.g., processors).
- Fairness
  - Resources apportioned fairly between threads

© Kavita Bala, Computer Science, Cornell University

Two threads, one counter

Web servers use concurrency:

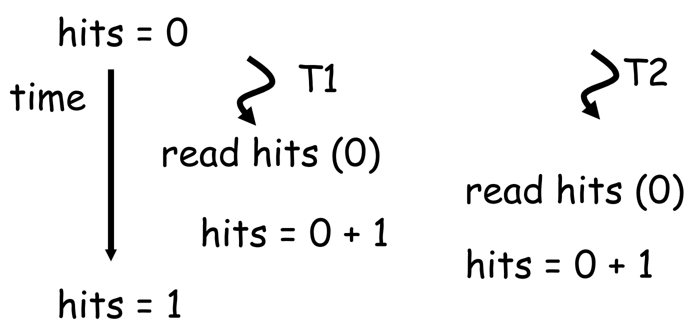
- Multiple threads handle client requests in parallel.
- Some shared state, e.g. hit counts:
  - each thread increments a shared counter to track number of hits

```
…
hits = hits + 1;
…
```

- What happens when two threads execute concurrently?

Shared counters

- Possible result: lost update!



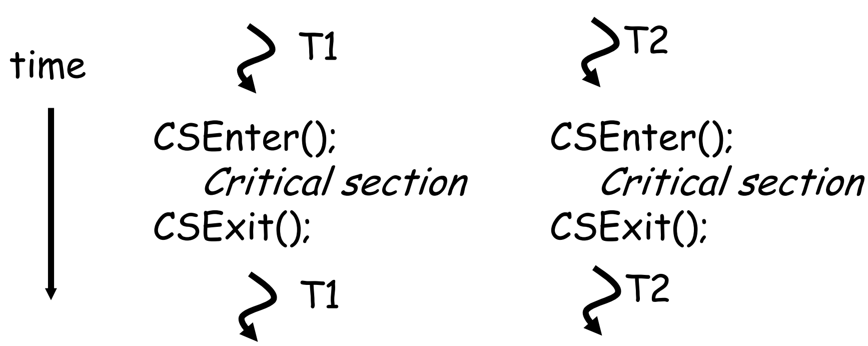
- Timing-dependent failure ⇒ race condition
  - hard to reproduce ⇒ Difficult to debug

Race conditions

- Def: timing-dependent error involving access to shared state
  - Whether it happens depends on how threads scheduled: who wins “races” to instruction that updates state vs. instruction that accesses state
  - Races are intermittent, may occur rarely
    - Timing dependent = small changes can hide bug
  - A program is correct *only* if *all possible* schedules are safe
    - Number of possible schedule permutations is huge
    - Need to imagine an adversary who switches contexts at the worst possible time

Critical sections

- To eliminate races: use *critical sections* that only one thread can be in
  - Contending threads must wait to enter



© Kavita Bala, Computer Science, Cornell University

Mutexes

- Critical sections typically associated with mutual exclusion locks (*mutexes*)
- Only one thread can hold a given mutex at a time
- Acquire (lock) mutex on entry to critical section
  - Or block if another thread already holds it
- Release (unlock) mutex on exit
  - Allow one waiting thread (if any) to acquire & proceed

```
pthread_mutex_init(m);
```

T1	T2
`pthread_mutex_lock(m);`	`pthread_mutex_lock(m);`
`hits = hits+1;`	`hits = hits+1;`
`pthread_mutex_unlock(m);`	`pthread_mutex_unlock(m);`

Using atomic hardware primitives

- Mutex implementations usually rely on special hardware instructions that *atomically* do a read and a write.
- Requires special memory system support on multiprocessors

Mutex init: lock = false;

```
while (test_and_set(&lock));

Critical Section

lock = false;
```

test_and_set uses a special hardware instruction that sets the lock and returns the OLD value (true: locked; false: unlocked)
- Alternative instruction: compare & swap, load linked/store conditional
- on multiprocessor/multicore: expensive, needs hardware support

Happy Thanksgiving!