

CS 2

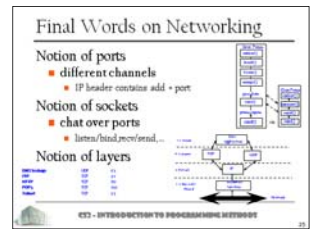
Introduction to Programming Methods



Last Time

Networking and Graph Algorithms

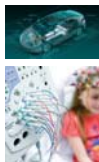
- Dijkstra and all...



Today's Topic

Concurrency

- designing and understanding systems that use parallel processing
 - virtually *all* systems in use today use parallel processing, including most safety-critical systems used in cars, airplanes, and medical devices
- touches on fundamental issues
 - predictable *performance*, functional *correctness*
 - of multicore processors, supercomputers, etc.
- just an intro
 - see G. Holzmann's class(es) and CS24 for more



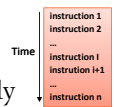
What is Concurrency?

Dealt with sequential algorithms only

- can point to code & say "the program is here"
 - single "thread" of execution; Program Counter

A bit simplistic...

- many programs happen concurrently
 - Word, Photoshop, email, music, ... (called processes)
- even within a program, multiple "threads"
 - Word: interpreting keystrokes, formatting line and page breaks, spell checking...



Real vs. Simulated Parallelism

Computations happen at same physical time

- on different computers
- on multiple processors of a same computer
- on multiple cores of a single processor

Computations *seem* to happen at same time

- when OS/program switches rapidly between different *programs/threads*

Fine and dandy if no shared resources...

- concurrency issues otherwise

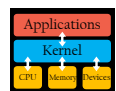
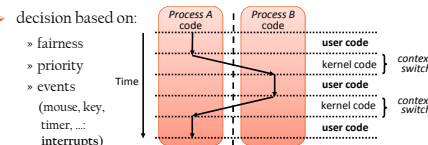
Switching between Processes

Processes managed by *kernel*

- piece of OS code handling access to CPU

- goes from one process to another
 - to use CPU time most efficiently...
 - and be fair to all processes (time sharing)

- via a *context switch*



Example of Concurrency Issues

Assume shared memory

- processes can read/write the same variable

Start $f()$ and $g()$ at the “same” time (and $x=0$)

```
f():
global x
for i=1 to 1000
  x = x + 1
```

```
g():
global x
for i=1 to 1000
  x = x + 1
```

- “ $x=x+1$ ” means read memory, add 1, write to memory
- remember, execution will lead to intertwined code lines
- question: what is x when all is finished?

Execution Order

Suppose $x = 43$
at a certain point



Net result: lost an increment

- called a **race condition** (non-deterministic results)
- hard to detect: problems are intermittent
 - all possible schedules have to be safe
 - but number of possible schedule permutations is huge!

Safe way to increment x without issues?

Synchronization

Concurrently-executing threads/processes should not execute **specific portions of a program** at the same time

- notion of **critical sections (CS)**
 - $x=x+1$ should be done atomically
- could make special hardware instructions
 - but does not extend to database systems or web servers
- need software solutions
 - used by the coder, or/and by the kernel

Critical Section Requirements

Mutual Exclusion

- only one process is in CS at any time

Progress

- not every process waiting for CS indefinitely
- no deadlock

Bounded Waiting

- bounded #times others get CS while one is waiting
- no livelock

When two trains approach each other at a crossing, both shall come to a full stop and neither shall start up again until the other has gone.
Statute passed by Kansas Legislature

Also: fairness, simplicity/efficiency, ...

Taking Turn?

Use a variable to determine turn

```
Process P0:
repeat
  while(turn!=0){};
  CS
  turn:=1;
  RS
forever

Process P1:
repeat
  while(turn!=1){};
  CS
  turn:=0;
  RS
forever
```

- achieves Mutual Exclusion (thru busy wait)
- but progress requirement is **not** satisfied
 - it requires strict alternation of CS's.
 - If a process asks for CS more often than the other, it can't get it

Checking Intent?

Use 2 intent flags

Initialization:
flag[0]:=flag[1]:=false

```
Process P0:
repeat
  flag[0]:=true;
  while(flag[1]){};
  CS
  flag[0]:=false;
  RS
forever

Process P1:
repeat
  flag[1]:=true;
  while(flag[0]){};
  CS
  flag[1]:=false;
  RS
forever
```

- Mutual Exclusion is satisfied
- but not the progress requirement
 - for the (interleaved) sequence:
 - flag[0]:=true
 - flag[1]:=true
 - both processes will wait forever (deadlock)

Peterson's Solution

Tie-breaking

- Even if both flags go up
 - and no matter how the instructions are interleaved
- ... turn will always end up as either 0 or 1

```

Initiation:
flag[0]=flag[1]=false
turn= 0 or 1

Process P0:
repeat
  flag[0]=true;
  // 0 wants in
  turn:= 1;
  // 0 gives a chance to 1
  while (flag[1] & turn==1);
  CS
  flag[0]=false;
  // 0 is done
  RS
forever

Process P1:
repeat
  flag[1]=true;
  // 1 wants in
  turn:=0;
  // 1 gives a chance to 0
  while (flag[0] & turn==0);
  CS
  flag[1]=false;
  // 1 is done
  RS
forever
    
```

Correctness

Mutual exclusion holds since:

- both P_0 and P_1 in CS means
 - both $\text{flag}[0]$ and $\text{flag}[1]$ must be true...
 - and $\text{turn}=0$ and $\text{turn}=1$ (at same time): impossible

Progress

- P_0 can be kept out of CS only if stuck in while loop
 - $\text{flag}[1] = \text{true}$ and $\text{turn} = 1$.
- If P_1 not ready to enter CS, $\text{flag}[1] = \text{false} \rightarrow P_0$ can enter CS
- If P_1 has set $\text{flag}[1]$, it is also in its while loop; then either P_0 or P_1 will go depending on value of turn
- progress condition is met

```

Process P0:
repeat
  flag[0]=true;
  // 0 wants in
  turn:= 1;
  // 0 gives a chance to 1
  while (flag[1] & turn==1);
  CS
  flag[0]=false;
  // 0 is done
  RS
forever
    
```

Correctness

Bounded Wait?

- suppose P_0 gets to go this time;
 - Can it go a second time indept of what P_1 does?
 - if P_1 enters CS, then $\text{turn}=1$
 - but will then reset $\text{flag}[1]$ to false on exit
 - thus allowing P_0 to enter CS
 - what if P_1 tries again, and has time to reset $\text{flag}[1]=\text{true}$ before P_0 gets to its CS?
 - it must also set $\text{turn}=0$
 - since P_0 is (stuck) past the point where it sets turn to 1;
 - P_0 will get to enter CS after at most one CS entry by P_1

```

Process P0:
repeat
  flag[0]=true;
  // 0 wants in
  turn:= 1;
  // 0 gives a chance to 1
  while (flag[1] & turn==1);
  CS
  flag[0]=false;
  // 0 is done
  RS
forever
    
```

Semaphores

Introduced by Dijkstra (1968)

- higher-level tool for process synchronization
 - uses an integer variable s ...
 - initialized with # of available resources (think: room reservation; licence counts)
 - and two operations:
 - $P(s)$:
 - atomically executes $s := s - 1$
 - until there's an opening
 - delays until $s > 0$
 - $V(s)$:
 - atomically executes $s := s + 1$
- binary semaphore = mutex
 - $P(s)$ before CS, $V(s)$ after CS

No Busy Waiting!

OS semaphore w/ wait queue

- to put process to sleep

```

void P(Semaphore *S) {
  S->value = S->value - 1;
  if (S->value < 0) {
    add this process to S.L;
    block();
  }
}
    
```

```

void V(Semaphore *S) {
  S->value = S->value + 1;
  if (S->value <= 0) {
    remove process P from S.L;
    wakeup(P)
  }
}
    
```

- $\text{block}()$: remove process from active pool
- $\text{wakeup}()$: back in kernel pool and resume

queue is FIFO
or priority based

```

typedef struct semaphore {
  int value;
  SleepingProcessesList L;
} Semaphore;
    
```

Possible Use of Concurrency

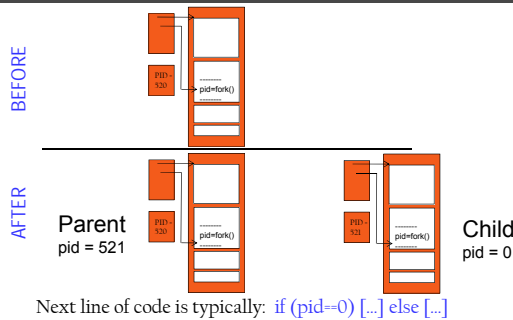
Many recursive methods involve 2 calls

- e.g., merge sort: left half and right half
- can be co-invoked!

Creation of threads by, e.g., fork/join

- $\text{fork}()$ spawns copy

Forking



CS2 - INTRODUCTION TO PROGRAMMING METHODS

19

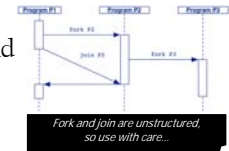
Possible Use of Concurrency

Many recursive methods involve 2 calls

- e.g., merge sort: left half and right half
- can be co-invoked!

Creation of threads by, e.g., fork/join

- fork() spawns copy
- join() waits for it to end



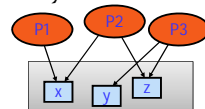
CS2 - INTRODUCTION TO PROGRAMMING METHODS

20

Communication & Synchronization

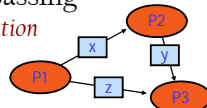
We looked at the shared memory model

- proc. "communicate" thru shared variables
- requires synchronization for memory access



Other approach: message-passing

- communication & synchronization are combined
- (a)synchronous comm.



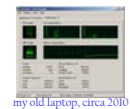
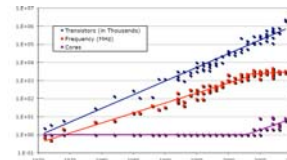
CS2 - INTRODUCTION TO PROGRAMMING METHODS

21

Why Should You Care?

Parallel computing increasingly important

- was trendy in the 70's and 80's
- but *really* necessary now... why?
- Moore's law, with a twist

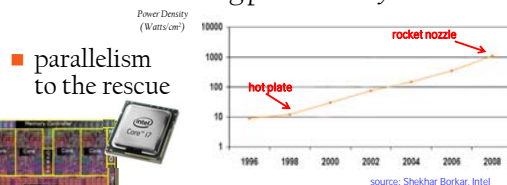


CS2 - INTRODUCTION TO PROGRAMMING METHODS

22

Reason: Too Hot...

Increasing transistor density...
... plus increasing clock-speed ...
means increasing *power density*



CS2 - INTRODUCTION TO PROGRAMMING METHODS

23