CPSC 457
Operating Systems

Lecture 9

Page Replacement Algorithms

Concurrency

Last Time

Virtual Memory

- Fragmentation
- Paging
- Page Tables
- Translation Look-aside Buffers
- Page Faults
- Locality of Reference

This Time

Virtual Memory

- Page Replacement Algorithms
- Thrashing
- Kernel Memory Management
- Memory Management Considerations
- Real World Memory Management

Concurrency

- Race Conditions (and other concurrent problems)
- Critical Sections
- Atomic Operations
- Locks
  - Mutexs
  - Semaphores

What to do when our frames are full?

Page Replacement Algorithms

Optimal

FIFO

Second Chance

Not Recently Used

Least Recently Used

Working Set

Optimal

First In, First Out

Second Chance

Not Recently Used

Least Frequently Used

Working Set

Thrashing

Kernel Memory Allocation

Buddy Allocation

SLAB Allocation

Memory Management Considerations

- **Local vs Global Allocation**
- **Non-Uniform Memory Access**
- **Page Size Selection**
- **How we code**
- **Page Locking**

Real World Memory Allocation

Windows

- Clustering / Working Set

Linux

- LRU – With a clock

Concurrency

What are the consequences of running processes at the same time?

What can we do to make sure that processes don't run over each other, while keeping the benefits of parallelism.

Race Conditions

What is the value of counter?

```
A: register1 = counter
B: register1 = register1 + 1          P1
C: counter = register1
```

```
D: register2 = counter
E: register2 = register2 - 1          P2
F: counter = register2
```

Critical Section

An area of code where processes change something common

1. Mutual Exclusion. No two processes may be simultaneously inside their critical regions
2. Universality. No assumptions may be made about speeds or numbers of CPUs
3. Progress. No process running outside its critical region may block any process
4. Bounded Waiting. No process should have to wait forever to enter its critical region

Atomic Operations

`Test_and_Set_Lock`

Simultaneously take the value of a variable and set it

Locks

Mutexes

`Lock`

Take the lock if available, wait if not

`Unlock`

Free the lock and wake the waiting processes

Semaphores

`Wait`

Wait for access to a resource

`Signal`

Let waiting processes know to proceed

Next Time

Concurrency

- Classical problems
  - Producer / Consumer
  - Dining Philosophers
  - Sleeping Instructor

Deadlock

- How locks can break execution