CSE 451: Operating Systems

Section 7:
Project 2b; Virtual Memory

Project 2a grading

* Grades will be sent out later tonight

Debugging threaded programs

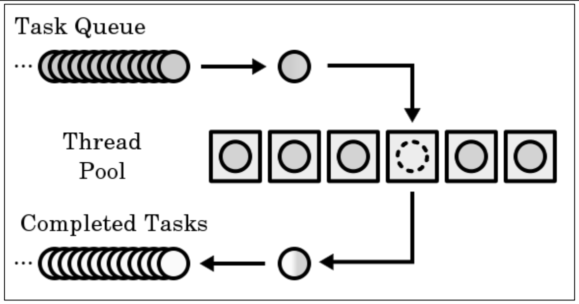
* What techniques have you used?
* printf statements: macros are helpful

```
#define print_debug(f, a...)  do {         \
  fprintf(stdout, "DEBUG: %lu: %s: " f,   \
          pthread_self(), __func__, ##a); \
  fflush(stdout);                         \
  } while(0)
```


Debugging threaded programs

* Other tools:
  * gdb
    * Deadlock vs. corruption
    * To enable core dumps: **ulimit -c unlimited**
  * helgrind? DRD?
  * MS Visual Studio; Intel Inspector XE; …
* What does the textbook say?
* We've mostly discussed thread correctness; what about thread *performance*?

Thread pools



- What is the “type” of a task?
- What function do the threads run?
- Can we make this abstract / generic?

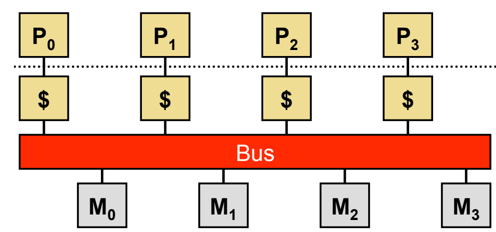
sioux thread pool

```
struct thread_pool {
   queue request_queue;
   sthread_cond_t request_ready;
};

struct request {
   int next_conn;
};


// New request arrives:
//   enqueue request, signal request_ready
// Worker threads:
//   dequeue, run: handle_request(request);
```


Generic thread pool

```
struct thread_pool {
   queue task_queue;
   sthread_cond_t work_to_do;
};
typedef void (*work_fn) (void *);
struct task {
   work_fn work;
   void *arg;
};

// New work arrives:
//   enqueue new task, signal work_to_do
// Worker threads:
//   dequeue, run: task->work(task->arg);
```


Thread *performance*



- Where might there be performance bottlenecks with a thread pool?
  - Where are threads running?
  - What do threads have to do to access thread pool?
  - Where is the work queue stored?

Image from: http://www.cis.upenn.edu/~milom/cis501-Fall09/lectures/09_multicore.pdf

Synchronization is expensive

* Explicit synchronization
  * Critical sections protected by mutexes, condition variables and queues
  * Strategies: reduce critical section size; atomic updates / lock-free data structures; RCU; …
* Implicit synchronization
  * Through cache coherence / memory hierarchy
  * Strategies: partitioning / sharding

Debugging thread performance

* How can we debug thread performance?
  * Intuition?
  * Profiling tools:
    * cachegrind / callgrind
    * Intel VTune Amplifier

sioux web server

* Make the web server multithreaded
  * Create a thread pool
    * Suggestion: create separate thread_pool.h, thread_pool.c
  * Wait for a connection
  * Find an available thread to handle the request
    * Request waits if all threads busy
  * Once the request is handed to a thread, it uses the same processing code as before
* Use pthreads for parts 4 and 6: we won't test sioux with sthreads!

Preemption (part 5)

* Remember this tip from previous section:
  * One way to think about preemption-safe thread library:
    * Disable/enable interrupts in "library" context
    * Use atomic locking in "application" context
* Does locking / unlocking a mutex happen in "library context" or "application context"?

How *not* to implement mutexes

```
sthread_user_mutex_lock(mutex)
   splx(HIGH);
   if (mutex->held) {
      enqueue(mutex->queue, current_thread);
      schedule_next_thread();
   } else {
      mutex->held = true;
   }
   splx(LOW);
```


How *not* to implement mutexes

* Don't turn it into a spinlock:

```
sthread_user_mutex_lock(mutex)
  while(atomic_test_and_set(
                     &(mutex->available))) { }
```

* This is also wrong: where could we get preempted that could lead to deadlock?

```
sthread_user_mutex_lock(mutex)
  while(atomic_test_and_set(
                       &(mutex->available))) {
     enqueue(mutex->queue, current_thread);
     schedule_next_thread();
   }
```

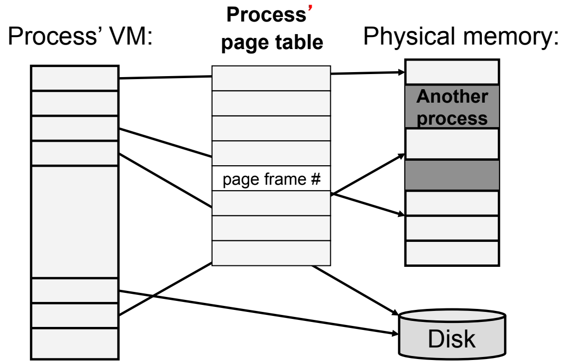

So how *does* one implement mutexes?

* Need to lock around the critical sections in the mutex functions themselves!
  * Your `struct _sthread_mutex` will likely need another member for this
* For hints, re-read lecture slides:
  * Module 7: Synchronization (slide 20 forward)
  * Module 8: Semaphores
* Similar hints apply for condition variables

Project 2b

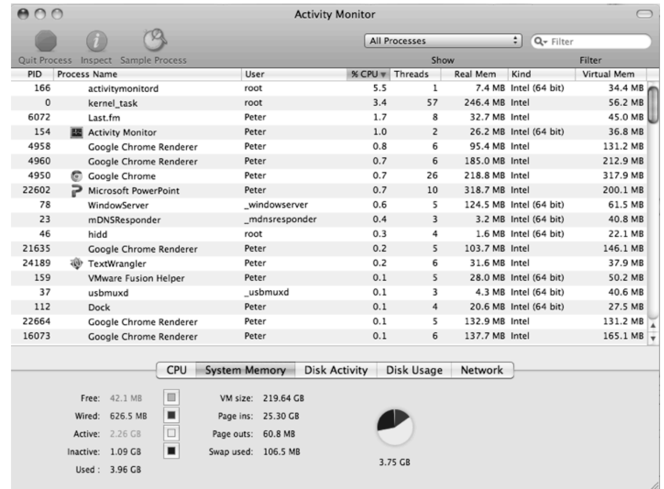
* Any more questions?

Virtual memory



How can we use paging to set up sharing of memory between two processes?

(slides from Chernyak Fall 2009)



Page replacement algorithms

* FIFO (First in/first out)
  * Replace the oldest page with the one being paged in
  * Not very good in practice, suffers from Belady's Anomaly
* Second-Chance (Modified FIFO)
  * FIFO, but skip referenced pages
  * VAX/VMS used this
* Random
  * Better than FIFO!
* NFU (Not Frequently Used)
  * Replace the page used the least number of times
  * Better variation: Aging ensures that pages that have not been used for a while go away.
* NRU (Not Recently Used)
  * Replace a page not used since last clock cycle
* LRU (Least Recently Used)
  * Replace the least recently used page
  * Works well but expensive to implement. (More efficient variants include LRU-K)
* LRU Clock (Modified LRU)
  * Replace the least recently used page, with a hard limit on the max time since used
* Clairvoyant
  * Replace the page that's going to be needed farthest in the future.

Example of Belady's anomaly

Sequence of page requests:	3	2	1	0	3	2	4	3	2	1	0	4
3 physical page frames:	**3**	3	3	**0**	0	0	**4**	4	4	4	4	4
		2	2	2	**3**	3	3	3	3	**1**	1	1
Page faults (in red): 9			**1**	1	1	**2**	2	2	2	2	**0**	0

Example of Belady's anomaly

Sequence of page requests:	3	2	1	0	3	2	4	3	2	1	0	4
4 physical page frames:	3	3	3	3	3	3	4	4	4	4	0	0
		2	2	2	2	2	2	3	3	3	3	4
			1	1	1	1	1	1	2	2	2	2
Page faults (in red): 10				0	0	0	0	0	0	1	1	1

5/10/12 21