CS194-24
Advanced Operating Systems Structures and Implementation
Lecture 6

Processes(con't), IPC, Scheduling Parallelism and POSIX Threads

February 9th, 2014

Prof. John Kubiatowicz

http://inst.eecs.berkeley.edu/~cs194-24

Goals for Today

- Processes, Scheduling (First Take)
- Interprocess Communication
- Multithreading
- Posix support for threads

Interactive is important!
Ask Questions!

Note: Some slides and/or pictures in the following are adapted from slides ©2013

Recall: example of *fork( ): Web Server*

Serial Version

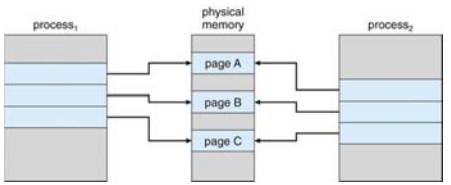
```
int main() {
  int listen_fd = listen_for_clients();
  while (1) {
    int client_fd = accept(listen_fd);
    handle_client_request(client_fd);
    close(client_fd);
  }
}
```

Process Per Request

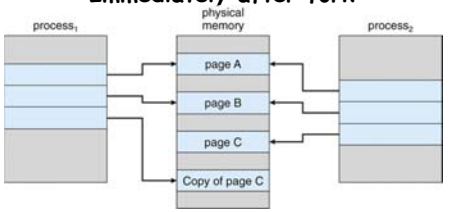
```
int main() {
  int listen_fd = listen_for_clients();
  while (1) {
    int client_fd = accept(listen_fd);
    if (fork() == 0) {
      handle_client_request(client_fd);
      close(client_fd);   // Close FD in child when done
      exit(0);
    } else {
      close(client_fd);   // Close FD in parent

      // Let exited children rest in peace!
      while (waitpid(-1,&status,WNOHANG) > 0);
    }
}
```


Recall: How does fork() actually work in Linux?

- Semantics are that fork() duplicates a lot from the parent:
  - Memory space, File Descriptors, Security Context
- How to make this cheap?
  - Copy on Write!
  - Instead of copying memory, copy page-tables
- Another option: vfork()
  - Same effect as fork(), but page tables not copied
  - Instead, child executes in parent's address space Parent blocked until child calls "exec()" or exits
  - Child not allowed to write to the address space



Immediately after fork

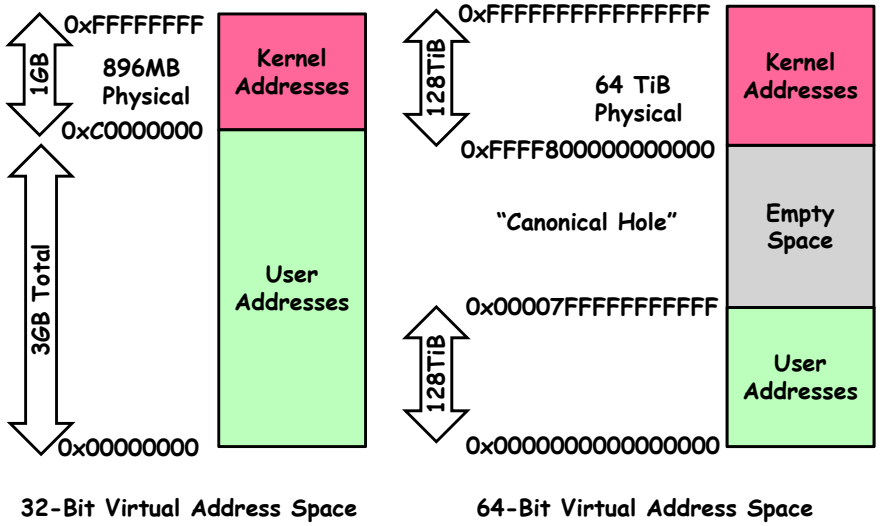


After Process-1 writes Page C

Some preliminary notes on Kernel memory

- Many OS implementations (including linux) map parts of *kernel* into every address space
  - Allows *kernel* to see into client processes
    - » Transferring data
    - » Examining state
- In Linux, Kernel memory not generally visible to user
  - Exception: special VDSO facility that maps kernel code into user space to aid in system calls (and to provide certain actual system calls such as gettimeofday().
- Every physical page described by a "page" structure
  - Collected together in lower physical memory
  - Can be accessed in kernel virtual space
  - Linked together in various "LRU" lists
- For 32-bit virtual memory architectures:
  - When physical memory < 896MB
    - » All physical memory mapped at 0xC0000000
  - When physical memory >= 896MB
    - » Not all physical memory mapped in kernel space all the time
    - » Can be temporarily mapped with addresses > 0xCC000000
- For 64-bit virtual memory architectures:
  - All physical memory mapped above 0xFFFF800000000000

Linux Virtual memory map



Physical Memory in Linux

- Memory Zones: physical memory categories
  - ZONE_DMA: < 16MB memory, DMAable on ISA bus
  - ZONE_NORMAL: 16MB ⇒ 896MB (mapped at 0xC0000000)
  - ZONE_HIGHMEM: Everything else (> 896MB)
- Each zone has 1 freelist, 2 LRU lists (Active/Inactive)
  - Will discuss when we talk about memory paging
- Many different types of allocation
  - SLAB allocators, per-page allocators, mapped/unmapped
- Many different types of allocated memory:
  - Anonymous memory (not backed by a file, heap/stack)
  - Mapped memory (backed by a file)
- Allocation priorities
  - Is blocking allowed/etc

Kmalloc/Kfree: The "easy interface" to memory

- Simplest kernel interface to manage memory: `kmalloc()/kfree()`
  - Allocate chunk of memory in kernel's address space (will be physically contiguous and virtually contiguous):

```
void * kmalloc(size_t size, gfp_t flags);
```

  - Example usage:

```
struct dog *p;
p = kmalloc(sizeof(struct dog), GFP_KERNEL);
if (!p)
        /* Handle error! */
```

  - Free memory: `void kfree(const void *ptr);`
  - Important restrictions!
    - » Must call with memory previously allocated through `kmalloc()` interface!!!
    - » Must not free memory twice!
- Alternative: `vmalloc()/vfree()`
  - Will be only contiguous in virtual address space
  - No guarantee contiguous in physical RAM
    - » However, only devices typically care about difference
  - `vmalloc()` may be slower than `kmalloc()`, since kernel must deal with memory mappings

Allocation flags

- Possible allocation type flags:
  - GFP_ATOMIC: Allocation high-priority and must never sleep. Use in interrupt handlers, top halves, while holding locks, or other times cannot sleep
  - GFP_NOWAIT: Like GFP_ATOMIC, except call will not fall back on emergency memory pools. Increases likely hood of failure
  - GFP_NOIO: Allocation can block but must not initiate disk I/O.
  - GFP_NOFS: Can block, and can initiate disk I/O, but will not initiate filesystem ops.
  - GFP_KERNEL: Normal allocation, might block. Use in process context when safe to sleep. This should be default choice
  - GFP_USER: Normal allocation for processes
  - GFP_HIGHMEM: Allocation from ZONE_HIGHMEM
  - GFP_DMA Allocation from ZONE_DMA. Use in combination with a previous flag

More sophisticated memory allocation (more later)

- One (core) mechanism for requesting pages: everything else on top of this mechanism:
  - Allocate contiguous group of pages of size 2^{order} bytes given the specified mask:

```
struct page * alloc_pages(gfp_t gfp_mask,
                          unsigned int order)
```

  - Allocate one page:

```
struct page * alloc_page(gfp_t gfp_mask)
```

  - Convert page to logical address (assuming mapped):

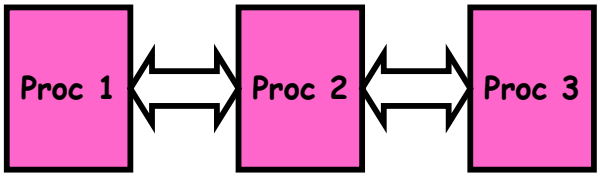
```
void * page_address(struct page *page)
```

- Also routines for freeing pages
- Zone allocator uses "buddy" allocator that tries to keep memory unfragmented
- Allocation routines pick from proper zone, given flags

Administrivia

- Groups posted on Website and in Redmine
  - See "Group Membership" link
  - Problems with infrastructure? Let us know!
- Piazza Usage:
  - Excessive posts may not be useful
  - Before asking someone:
    - » Try Google or "man"
    - » Read the code!
    - » Can get function prototypes by looking at header files
- Developing FAQ – please tell us about problems
- Design Document
  - What is in Design Document?
  - BDD => "No Design Document"!?

Multiple Processes Collaborate on a Task



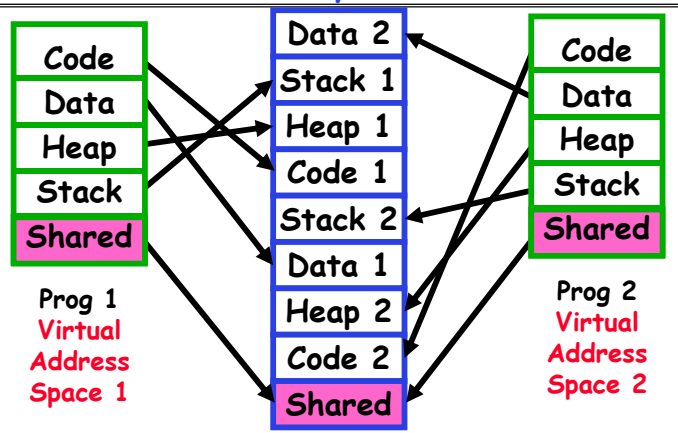
- (Relatively) High Creation/memory Overhead
- (Relatively) High Context-Switch Overhead
- Need Communication mechanism:
  - Separate Address Spaces Isolates Processes
  - Shared-Memory Mapping
    - » Accomplished by mapping addresses to common DRAM
    - » Read and Write through memory
  - Message Passing
    - » `send()` and `receive()` messages
    - » Works across network
  - Pipes, Sockets, Signals, Synchronization primitives, ….

Message queues

- What are they?
  - Similar to the FIFO pipes, except that a tag (type) is matched when reading/writing
    - » Allowing cutting in line (I am only interested in a particular type of message)
    - » Equivalent to merging of multiple FIFO pipes in one
- Creating a message queue:
  - int msgget(key_t key, int msgflag);
  - Key can be any large number. But to avoiding using conflicting keys in different programs, use ftok() (the key master).
    - » key_t ftok(const char *path, int id);
      - Path point to a file that the process can stat
      - Id: project ID, only the last 8 bits are used
- Message queue operations

  int msgget(key_t, int flag)
  int msgctl(int msgid, int cmd, struct msgid_ds *buf)
  int msgsnd(int msgid, const void *ptr, size nbytes, int flag);
  int msgrcv(int msgid, void *ptr, size_t nbytes, long type, int flag);
- Performance advantage is no longer there in newer systems (compared with pipe)

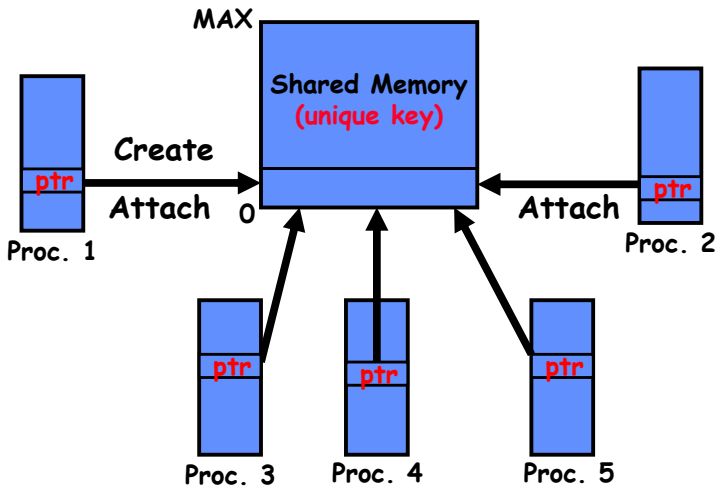
Shared Memory Communication



- Communication occurs by "simply" reading/writing to shared address page
  - Really low overhead communication
  - Introduces complex synchronization problems

Shared Memory

Common chunk of read/write memory among processes



Creating Shared Memory

```
// Create new segment
int shmget(key_t key, size_t size, int shmflg);
```

Example:

```
key_t key;
int shmid;

key = ftok("<somefile>", 'A');

shmid = shmget(key, 1024, 0644 | IPC_CREAT);

Special key: IPC_PRIVATE (create new segment)
Flags: IPC_CREAT (Create new segment)
       IPC_EXCL (Fail if segment with key already exists)
       lower 9 bits – permissions use on new segment
```

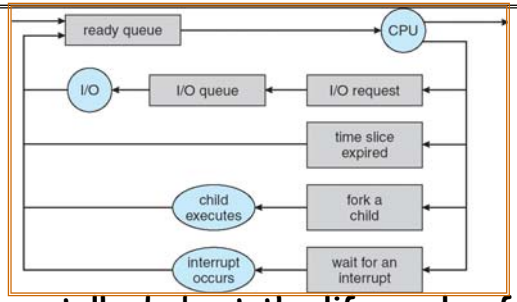
Attach and Detach Shared Memory

```
// Attach
void *shmat(int shmid, void *shmaddr, int shmflg);
// Detach
int shmdt(void *shmaddr);
```

Example:

```
key_t key;
int shmid;
char *data;

key = ftok("<somefile>", 'A');
shmid = shmget(key, 1024, 0644);
data = shmat(shmid, (void *)0, 0);

shmdt(data);

Flags: SHM_RDONLY, SHM_REMAP
```


Recall: CPU Scheduling

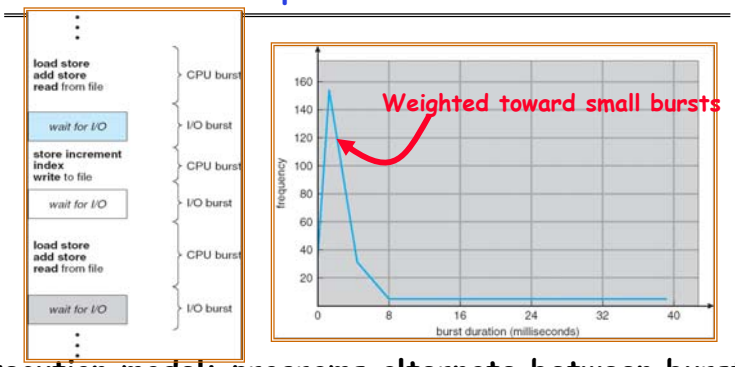


- Earlier, we talked about the life-cycle of a thread
  - Active threads work their way from Ready queue to Running to various waiting queues.
- Question: How is the OS to decide which of several tasks to take off a queue?
  - Obvious queue to worry about is ready queue
  - Others can be scheduled as well, however
- Scheduling: deciding which threads are given access to resources from moment to moment

Scheduling Assumptions

- CPU scheduling big area of research in early 70's
- Many implicit assumptions for CPU scheduling:
  - One program per user
  - One thread per program
  - Programs are independent
- Clearly, these are unrealistic but they simplify the problem so it can be solved
  - For instance: is "fair" about fairness among users or programs?
    - » If I run one compilation job and you run five, you get five times as much CPU on many operating systems
- The high-level goal: Dole out CPU time to optimize some desired parameters of system

Assumption: CPU Bursts



- Execution model: programs alternate between bursts of CPU and I/O
  - Program typically uses the CPU for some period of time, then does I/O, then uses CPU again
  - Each scheduling decision is about which job to give to the CPU for use by its next CPU burst
  - With timeslicing, thread may be forced to give up CPU before finishing current CPU burst

Scheduling Policy Goals/Criteria

- Minimize Response Time
  - Minimize elapsed time to do an operation (or job)
  - Response time is what the user sees:
    - » Time to echo a keystroke in editor
    - » Time to compile a program
    - » Real-time Tasks: Must meet deadlines imposed by World
- Maximize Throughput
  - Maximize operations (or jobs) per second
  - Throughput related to response time, but not identical:
    - » Minimizing response time will lead to more context switching than if you only maximized throughput
  - Two parts to maximizing throughput
    - » Minimize overhead (for example, context-switching)
    - » Efficient use of resources (CPU, disk, memory, etc)
- Fairness
  - Share CPU among users in some equitable way
  - Fairness is not minimizing average response time:
    - » Better *average* response time by making system *less* fair

Example: First-Come, First-Served (FCFS) Scheduling

- First-Come, First-Served (FCFS)
  - Also "First In, First Out" (FIFO) or "Run until done"
    - » In early systems, FCFS meant one program scheduled until done (including I/O)
    - » Now, means keep CPU until thread blocks
- Example:

Process	Burst Time
P_1	24
P_2	3
P_3	3

  - Suppose processes arrive in the order: P_1 , P_2 , P_3 The Gantt Chart for the schedule is:

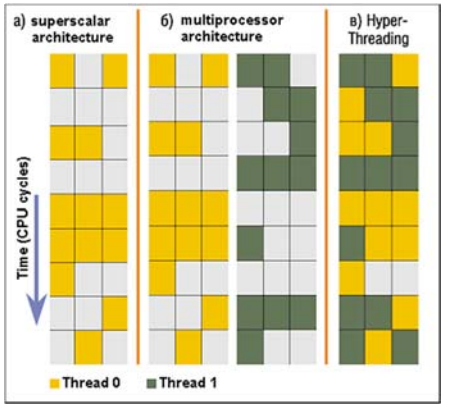
  - Waiting time for P_1 = 0; P_2 = 24; P_3 = 27
  - Average waiting time: (0 + 24 + 27)/3 = 17
  - Average Completion time: (24 + 27 + 30)/3 = 27
- *Convoy effect:* short process behind long process

Real-Time Scheduling Priorities

- Efficiency is important but predictability is essential
  - In RTS, performance guarantees are:
    - » Task- and/or class centric
    - » Often ensured a priori
  - In conventional systems, performance is:
    - » System oriented and often throughput oriented
    - » Post-processing (… wait and see …)
  - Real-time is about enforcing predictability, and does not equal to fast computing!!!
- Typical metrics:
  - Guarantee miss ratio = 0 (hard real-time)
  - Guarantee Probability(missed deadline) < X% (firm real-time)
  - Minimize miss ratio / maximize completion ratio (firm real-time)
  - Minimize overall tardiness; maximize overall usefulness (soft real-time)
- EDF (Earliest Deadline First), LLF (Least Laxity First), RMS (Rate-Monotonic Scheduling), DM (Deadline Monotonic Scheduling)

Recall: Modern Technique: SMT/Hyperthreading

- Hardware technique
  - Exploit natural properties of superscalar processors to provide illusion of multiple processors
  - Higher utilization of processor resources
- Can schedule each thread as if were separate CPU
  - However, not linear speedup!
  - If have multiprocessor, should schedule each processor first



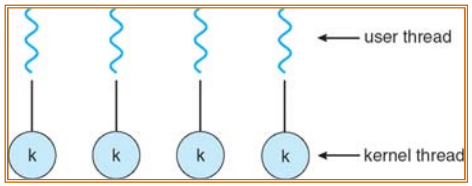
- Scheduling Heuristics?
  - Maximize pipeline throughput
  - Possibly include thread priority

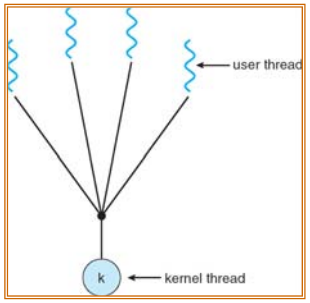
Kernel versus User-Mode threads

- We have been talking about Kernel threads (/tasks)
  - Native threads supported directly by the kernel
  - Every thread can run or block independently
  - One process may have several threads waiting on different things
- Downside of kernel threads for many OSes: a bit expensive
  - Need to make a crossing into kernel mode to schedule
  - Linux does make *creation* of threads pretty inexpensive
- Even lighter weight option: User Threads
  - User program provides scheduler and thread package
  - May have several user threads per kernel thread
  - User threads may be scheduled non-premptively relative to each other (only switch on yield())
- Downside of user threads:
  - When one thread blocks on I/O, all threads block
  - Kernel cannot adjust scheduling among all threads
  - Option: *Scheduler Activations*
    - » Have kernel inform user level when thread blocks…

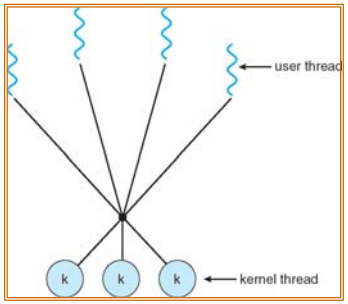
Threading models mentioned by Silbershatz book

Simple One-to-One Threading Model





Many-to-One



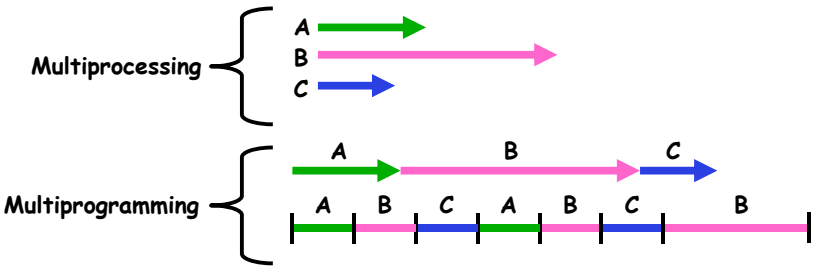
Many-to-Many

Thread Level Parallelism (TLP)

- In modern processors, Instruction Level Parallelism (ILP) exploits implicit parallel operations within a loop or straight-line code segment
- Thread Level Parallelism (TLP) explicitly represented by the use of multiple threads of execution that are inherently parallel
  - Threads can be on a single processor
  - Or, on multiple processors
- Concurrency vs Parallelism
  - Concurrency is when two tasks can start, run, and complete in overlapping time periods. It doesn't necessarily mean they'll ever both be running at the same instant.
    - » For instance, multitasking on a single-threaded machine.
  - Parallelism is when tasks literally run at the same time, eg. on a multicore processor.
- Goal: Use multiple instruction streams to improve
  - Throughput of computers that run many programs
  - Execution time of multi-threaded programs

Multiprocessing vs Multiprogramming

- Remember Definitions:
  - Multiprocessing ≡ Multiple CPUs
  - Multiprogramming ≡ Multiple Jobs or Processes
  - Multithreading ≡ Multiple threads per Process
- What does it mean to run two threads "concurrently"?
  - Scheduler is free to run threads in any order and interleaving: FIFO, Random, …
  - Dispatcher can choose to run each thread to completion or time-slice in big chunks or small chunks



Correctness for systems with concurrent threads

- If dispatcher can schedule threads in any way, programs must work under all circumstances
  - Can you test for this?
  - How can you know if your program works?
- Independent Threads:
  - No state shared with other threads
  - Deterministic ⇒ Input state determines results
  - Reproducible ⇒ Can recreate Starting Conditions, I/O
  - Scheduling order doesn't matter (if `switch()` works!!!)
- Cooperating Threads:
  - Shared State between multiple threads
  - Non-deterministic
  - Non-reproducible
- Non-deterministic and Non-reproducible means that bugs can be intermittent
  - Sometimes called "Heisenbugs"

Interactions Complicate Debugging

- Is any program truly independent?
  - Every process shares the file system, OS resources, network, etc
  - Extreme example: buggy device driver causes thread A to crash "independent thread" B
- You probably don't realize how much you depend on reproducibility:
  - Example: Evil C compiler
    - » Modifies files behind your back by inserting errors into C program unless you insert debugging code
  - Example: Debugging statements can overrun stack
- Non-deterministic errors are really difficult to find
  - Example: Memory layout of kernel+user programs
    - » depends on scheduling, which depends on timer/other things
    - » Original UNIX had a bunch of non-deterministic errors
  - Example: Something which does interesting I/O
    - » User typing of letters used to help generate secure keys

Summary

- Linux Memory Map
  - Portions of virtual memory reserved for kernel
    - » 32-bit machines: memory < 896 mapped at 0xC0000000
    - » 64-bit machines: all memory mapped at 0xFFFF800000000000
  - Kernel memory not necessarily visible to user in usermode
- Linux Kernel Memory Details
  - kmalloc()/kfree(): Kernel versions of normal malloc
  - alloc_pages(): Allocate pages directly from SLAB allocator
  - Flags specify terms of allocation, e.g.:
    - » GFP_KERNEL: Normal allocation, might block.
    - » GFP_USER: Normal allocation for process space
- Interprocessor Communication:
  - Message Queues/Shared Memory/Others
- Scheduling
  - Non-Realtime Goals: Throughput, Latency, Fairness
  - Realtime Goals: Meet deadlines, other QoS constraints