Processes (Chapt 3)

Early computers: Load program into memory, run

- Only one program in memory at a time
- No sharing
- Program in total control of all elements of the machine
- No security ...
- (Assignment 1 does this on qemu/RISCV)

Shared computers:

- Have multiple programs "running" at the same time
  - called multiprogramming, sharing CPU
  - With multi-CPU machines, concurrent processes
- A process shouldn’t "interfere" with others
- A process shouldn’t "see" unrelated processes (not always enforced)
- A group of processes should be allowed to "work together"

Abstraction of process (task, job, ...)

- Even useful on single user systems (e.g. moble devices)
- Process consists of:
  - executable file (program)
  - "memory image"
    - text, data, heap, stack
  - CPU state

Process "state"

- new: in the process of being created
- ready: ready to run, waiting on a CPU
- running: CPU actually running instructions
- waiting: "blocked" waiting an event
- terminated: finished execution, not cleaned up

- process state transitions (Sect 3.1.2)

Process Control Block

Kernel Data Structure -- keeps track of a process: Process table of Process Control Blocks

- State (last slide)
- Program Counter (PC)
- CPU Registers
- CPU Scheduling Information
- Memory Management Information
- Accounting Information (time used, PID, ...)
- I/O status information

With threading: multiple state/PC/Regs/Scheduling per thread

- Why not Memory Management?
  - Single memory image, multiple threads

Process Scheduling

What process (job) to run next?

- Kernel scheduler
  - policy: which one runs next
  - mechanism: switching processes (context switch)
- Queuing
  - Ready Queue: processes ready to run
  - Device/Event Queues: processes waiting on device/event
- Dispatch -- select a process for execution

Scheduler:

- part of the OS that makes the policy decision
- small amount of code
- controls degree of multiprogramming (number of processes in memory)
- can have large impacts on system performance
  - e.g. swapping -- moving entire processes memory image to disk
  - I/O bound processes get selected first?
  - Compute bound processes get selected first?
  - Process Mix: Priority vs Round Robin vs ...

Process Creation

Two Primary methods

- Clone (e.g. UNIX -- fork())
  - copy a current process
  - new process running same program
  - running a new program is different system call (exec())

- Create New (e.g. Windows -- NtCreateProcess())
  - Creates process and specifies program at same system call
  - Very little is "inherited" from "parent"
  - Needs to specify a lot of things
  - Harder to simulate Clone with "Create New"
    - (Clone/exec can simulate Create New easily.)

Parent (Process creator) options

- Continues to run concurrently with children
- Wait for child to die
  - See code in book for UNIX and Windows versions

Process Termination

- Directly calling a routine to "exit"
- An error was detected and system "kills" the child
- Parent (or other process) can request system to "kill" child

Can a process become an orphan?

- UNIX -- Yes
- VMS -- No
  - A process terminates => kill all children
- Windows -- Yes

The first process ...

UNIX -- "init" started first, pid of 1.

Linux -- more recently started using a "systemd"

Interprocess Communication

Cooperating processes need to communicate ...

- Why cooperate?
  - Information sharing
  - Computational speedup
  - Modularity
  - Convenience

Two primary models

- Shared Memory
- Message Passing

Shared Memory systems

- System maps parts of both virtual memory space to same physical memory
- What is written by one process can be seen by all others sharing memory
- Brings up synchronization problems
  - Producer and Consumer problem
  - Unbounded buffer vs bounded buffer
- POSIX has specified a shared memory API

Interprocess Communication (page 2)

Message Passing Systems

- Pipes
- mail boxes
  - rendezvous -- both process must be in code at same time
  - buffered mail box -- asynchronous
- local sockets
- network sockets

Read the book for specific examples and code

Threads and Concurrency (Chapter 4)

Thread is the "basic unit of CPU utilization" consisting of:

- CPU State
  - Program Counter
  - Registers
  - Other information ... CPU dependent
- Stack

Process Threads --

- Traditional Heavyweight
- Multi-threaded
  - user level threads
  - kernel level threads

Why?

- Responsiveness
- Resource Sharing
- Economy
- Scalability
- Simulation
- Multi-core and systems with GPUs
  - task parallel vs data parallel (MIMD vs SIMD) (CSCI 415/515)

Thread Models

- Many to one (aka User Level Threads)
  - Kernel knows about only one thread
  - Library keeps track of threads
  - Block a thread blocks entire process
  - No concurrent running on a multi-processor
- One to one
  - Each user level thread has a kernel thread
  - Blocking one thread does not block other threads
  - Full concurrency on a multi-processor
  - Expensive in kernel resources
- Many to Many (aka M:N model)
  - Many user level threads
  - Fewer kernel threads
  - Not as expensive in kernel, still allows concurrency

Review Amdahl's Law (Sec 4.2.1) and "multicore programming"

Thread Issues in the OS

- Explicit threading: pthreads, windows threads, Java threads, ..
- Implicit threading: OpenMP, Grand Central Dispatch (Apple), ...
  - Both: Chapel!
- fork() and exec(), fork() duplicate all threads?
- thread cancellation
  - resource releasing
  - cancellation points
- signal delivery
  - to one thread vs to all threads
  - Often, delivered to a thread not blocking signal
- thread pools
- thread specific data
  - errno with concurrent system calls
  - user level per thread data (i.e. global to thread)

Read section 4.7

Synchronization Tools (Chapter 6) (Chapter 5 after this)

Race Condition (Review from CS 347)

- Results depend on the order of execution of the threads/processes.
- Book discusses the bounded buffer problem

Critical Section Solution (Requirements)

- Mutual exclusion -- no other process may be in critical section
- progress -- only processes wanting into critical section can participate in the selection of next process in critical section
- bounded waiting -- process gets critical section in a bounded manner

Peterson’s software solution

- Turn based, two processes only (numbered 0 and 1)
- flag[i] = TRUE; turn = j; while (flag[j] && turn == j) /*spin*/
- << critical section >>
- flag[i] = FALSE;

Synchronization Tools (page 2)

Hardware Solution

- Test and set done atomically by hardware (there are others)
  - If already set, still set.

```
while (TestAndSet(&lock)) /* spin */;
<< critical section >>
lock = 0;  /* or FALSE */
```

- Doesn’t solve bounded waiting ...
- Also does busy waiting (not that good to do)

Mutexes -- a help for critical sections

- Mutex -> Mutual Exclusion

```
mutex_lock(M)
<< critical section >>
mutex_unlock(M)
```

- Usually have some initialization done on the mutex
- no contention, low contention, high contention
- Implementation
  - Queue for FCFS
  - turn off interrupts on some systems
  - use of spin locks on multi-processor systems (yes, busy wait)

Synchronization Tools (page 3)

Semaphores:

- S is an integer
- wait(S) { while (S <= 0) /* wait */; S--; }
- signal(S) { S++; }
- These need to be atomic
- (Original P (Wait) for proberen, and V (Signal) for verhogen in Dutch)
- Critical region solution: Initialize S to 1
- wait(S); << critical Region >> ; signal(S)
  - Still doesn’t solve busy waiting or bounded waiting

Implementation to solve busy waiting and bounded waiting issues

- Semaphore: struct { int value; struct process *list; }

```
wait (semaphore *S)
  S->value --;
  if (S->value < 0) { add_to_list(self, S->list); block(); }
signal (semaphore *S)
  S->value++;
  if (S->value <= 0) { p = removefirst(S->list); wakeup(p); }
```

- Solves busy waiting and with a simple queue, solves unbounded waiting
- Block() and Wakeup() requires "OS help", wait/signal run in mutual exclusion

Synchronization Tools (page 4)

- Not used properly, semaphores can cause deadlocks
- P0: wait(S); wait(Q); .... signal(S); signal(Q);
- P1: wait(Q); wait(S); .... signal(Q); signal(S);

Monitors: Use of semaphores and mutexes can cause problems

- not using mutexes
- improper wait/signal sequences ...

Desire of language designers to "help" .... yielded monitors

- High level language abstraction
- ADT: only one thread may be executing inside a monitor at a time
  - shared data must be declared in the monitor
- Solves critical section, does not solve other issues
- Enter the "condition variable"
  - Wait -- put on a queue to be signaled
  - Signal -- if any process on the queue, let them run
    - no process on the queue ... do nothing
    - problem of signal and two processes running in monitor
    - solution: signal and leave.
- C++ monitors?
  - fair waiting

Other high level language constructs exist ... see Path Pascal

Read: sections 6.8 and 6.9

Classic Problems of Synchronization (Chapter 7)

- Bounded Buffer Problem (aka Producer-Consumer Problem): Fixed sized buffer: add(), remove()
  - Semaphores can help here
- Readers-writers problem: shared database
  - readers can run concurrently
  - writers must have exclusive access
  - writers can't be locked out long
    - block more readers when a writer wants to write
  - A monitor is a good implementation of this.
- Dining Philosophers

Why talk about these issues?

- Synchronization within the kernel!
- Each OS has to build synchronization primitives
  - Book talks about Windows and Linux
- POSIX defines mutexes, semaphores and condition variables
- Read the rest of chapter 7 ... won't talk about it here (7.4, 7.5)
- Assignment 2 has you implementing synchronization problems

CPU Scheduling (Chapter 5)

Basic requirement -- CPU Switch (aka dispatcher)

- a way to switch the CPU between processes
- Function (e.g. switch):
  - saves current CPU state -> PCB of current process
  - loads new CPU state <- PCB of new process
  - common to save registers on process stack
  - called from one "process", returns to next "process"

Job with CPU bursts

- Compute and I/O waits
- "wasted time" in wait
- multiprogramming to make use of that wait time

CPU Scheduler

- selects a process from the ready queue, does a switch
- ready queue may not be a true FIFO, e.g. processes may have priorities
- Nothing on the ready queue?
  - Idle process (wait for interrupt?)

Kinds of scheduling

- Run to completion
  - As long as the process needs the CPU, it gets it
  - Interrupts, timers ... are processed, but not other user processes
  - Processes can "yield" to others waiting
- Preemptive Scheduling
  - OS may "take the CPU" away from a process
  - Interrupts, timers get CPU back to OS
  - OS may then not give CPU back to currently running process

Scheduling criteria

- CPU Utilization -- keep CPU busy
- Throughput -- number of jobs finished
- Turnaround time -- time to completion for job
- Waiting time -- time spend waiting on the ready queue
- Response time -- on an interactive system

Scheduling algorithms

- First-Come, First Served
  - Convoy effect -- all short jobs wait for big jobs
- Shortest-Job-First
  - How to find out what is shortest?
    - previous CPU burst(s)
    - exponential average
- Priority Scheduling
  - Problem: starvation
    - Solution: aging ... over time increase priority
- Round Robin
  - Time quantum: use more -> preempted
  - shorter times favor response time
  - longer times favor getting more computing done

Scheduling algorithms (page 2)

- Multilevel Queue Scheduling
  - foreground and background queues
    - foreground 80% of CPU RR, background 20% FCFS
  - priority levels
    - Job class or "social level" e.g. student processes lowest
  - feedback back queues -- high priority, short quantum
    - use a full quantum, drop to next priority

What about scheduling for threads

- User Level Threads?
  - Library has to schedule
  - Library can be preemptive with signals
  - Kernel only schedules the one kernel level thread

Scheduling algorithms (page 3)

- Kernel Level Threads?
  - Process A with many threads vs Process B with one thread
    - Process based scheduling?
      - process-contention scope (PCS)
    - Thread based scheduling?
      - system-contention scope (SCS)
      - done by Windows, Solaris and Linux

- Multi-processor Scheduling?
  - Assuming homogeneous processors
  - Asymmetric multiprocessing
    - One CPU is master, does all scheduling decisions
    - Other CPUs just run user code
    - No "multi-processing" in OS

Scheduling algorithms (page 4)

Symmetric multiprocessing

- All CPUs run kernel code
- All CPUs make scheduling decisions
- Requires proper kernel thread coordination
  - don’t want same thread running on 2 or more CPUs.

Processor Affinity

- Instruction and Data caches
  - move thread to different CPU has to restart caching
- Possible special hardware on specific CPUs
- Multiple Layer Memory systems, NUMA (Non-uniform Memory Access)
  - Each CPU has fast link to some memory, slow to all other
- Hard affinity vs Soft Affinity

Load Balancing

- SMP - typically not a problem
- Run Queue per CPU => some may be busy others not
- Push or Pull migration
- Migration vs Processor Affinity
  - Migration defeats purpose of Affinity
- Larger MP systems ... e.g. MOSIX (now defunct)
  - Multi-system vs just Multi-CPU: fork() and forget ...

Multi-core processors Issues

- Single Data Path to Memory
- Memory Stall -- time CPU waits while accessing memory
- CPU schedules threads on cores
- Tries to overlap compute on one thread with memory stall on another thread

Read sections 5.5.5 to end of chapter

Deadlocks (Chapter 8)

multiprogramming environment: several processes may compete for a finite set of resources

Typical idea:

- Request a resource, if not available, wait for it.
- No progress if resource is not available

Problem:

- Proc A: Holds R1, Waits on R2
- Proc B: Holds R2, Waits on R1
- Deadlock!

Typical Resource use:

- Request : Use : Release
  - e.g. scanner

Deadlock conditions

- Mutual exclusion
- Hold and wait
- No preemption
- Circular wait

Deadlock (page 2)

Resource-Allocation graph can help one understand deadlock

- Set P -- Processes
- Set R -- Resources
- Directed Edges:
  - R_i -> P_j -- P_j holds resource R_i
  - P_i -> R_j -- P_i is waiting on R_j

- Cycle in a allocation graph => deadlock

Handling Deadlocks

- Ostrich method
- Deadlock Prevention
- Deadlock Avoidance
- Deadlock Detection

Ostrich method ?

- UNIX uses it!

Deadlock Prevention

Break one of the necessary conditions

- Mutual Exclusion?
  - Can’t ignore, there are sharable resources (e.g. read-only file)
  - Mutex lock -- protecting a read/write sharable resource
- Hold and Wait?
  - Request ALL at beginning?
  - Request when not holding?
  - Low resource utilization and possible starvation
- Preemption?
  - Take away a resource from a process to give to another
    - CPU? -- works well
    - Printer? -- not so good
- Circular Wait?
  - Request resources in the same order (R1, R2, ...)
  - Request resources so we are not holding any higher number R

Read about deadlock avoidance and recovery from deadlock .... we need to move on ...