Threads (cont.)

Pramook Khungurn

Thread

- A running module
- And all information that allow you to:
  - Stop it while it is running
  - Save this information somewhere
  - Resume the module later with the saved information
- The module has no idea that it was stopped and resumed.

Thread (cont.)

- Concretely, such information consists of:
  - Instruction pointer (or program counter)
  - Registers
    - Those used to do arithmetic calculations
    - Stack pointer (SP)
    - Page map address register (PMAR)
  - Other information:
    - Information about opened files
    - Information about CPU scheduling
    - Information about I/O

Last Time

- Non-preemptive scheduling
  - Threads agree to release the CPU periodically.
  - Nothing forces them to do this though.
  - **Soft modularity**
    - Threads share fate.
    - If one thread goes into an infinite loop, other threads cannot run.
- Need **preemptive scheduling** to enforce modularity.

Preemptive Scheduling

- Each thread is given a **time quantum** to run.
- Once it has used up the time quantum, the thread manager schedules another thread to run.
- Typically, a time quantum is 10-100 milliseconds.

Preemptive Scheduling (cont.)

- Needs an external mechanism to inform the thread manager that the time quantum has expired.
- The thread manager can’t do this by itself. (The CPU is being controlled by the running thread.)
- The external mechanism is the **clock interrupt**.
  – The thread manager can tell the clock circuit to fire an interrupt 100 millisecond from now.

Preemptive Scheduling (cont.)

- Note the difficulty:
  - Preemptive scheduling relies on **interrupts.**
  - Interrupts must be processed in kernel.
    - Can’t let user programs handle hardware directly.
  - What about preemptive scheduling in user programs?
- Let’s talk about preemptive scheduling in user programs later.
- Now, we’ll focus on preemptive scheduling of kernel threads.

Preemptive Scheduling of Kernel Threads

- All comes to handle clock interrupt.
- When an interrupt occurs, the CPU needs to do three things:
  - Save the states of the current kernel thread somewhere.
  - Change CPU mode to kernel.
  - Jump to the interrupt handler (specified in the interrupt vector).

Example: Interrupt in x86

- Can be triggered by:
  - Hardware --- “IRQ”
  - Software --- “trap” --- via INT instruction

x86 when an interrupt is fired

1. decide the vector number (depends on the source of the interrupt)
2. fetch the interrupt descriptor from the IDT.
3. check that CPL <= DPL in the descriptor (but only if INT instruction).
4. save ESP and SS in a CPU-internal register (but only if target segment selector's PL < CPL).
5. load SS and ESP from TSS ("")
6. push user SS ("")
7. push user ESP ("")
8. push user EFLAGS
9. push user CS
10. push user EIP
11. clear some EFLAGS bits
12. set CS and EIP from IDT descriptor's segment selector and offset

x86 when an interrupt is fired (cont.)

Figure out where the handler is.

1. decide the vector number (depends on the source of the interrupt)
2. fetch the interrupt descriptor from the IDT.
3. check that CPL <= DPL in the descriptor (but only if INT instruction).
4. save ESP and SS in a CPU-internal register (but only if target segment selector's PL < CPL).
5. load SS and ESP from TSS ("")
6. push user SS ("")
7. push user ESP ("")
8. push user EFLAGS
9. push user CS
10. push user EIP
11. clear some EFLAGS bits
12. set CS and EIP from IDT descriptor's segment selector and offset

x86 when an interrupt is fired (cont.)

1. decide the vector number (depends on the source of the interrupt)
2. fetch the interrupt descriptor from the IDT.
3. check that CPL <= DPL in the descriptor (but only if INT instruction).
4. save ESP and SS in a CPU-internal register (but only if target segment selector's PL < CPL).
5. load SS and ESP from TSS ("")
6. push user SS ("")
7. push user ESP ("")
8. push user EFLAGS
9. push user CS
10. push user EIP
11. clear some EFLAGS bits
12. set CS and EIP from IDT descriptor's segment selector and offset

Save states on stack of the current kernel thread

x86 when an interrupt is fired (cont.)

1. decide the vector number (depends on the source of the interrupt)
2. fetch the interrupt descriptor from the IDT.
3. check that CPL <= DPL in the descriptor (but only if INT instruction).
4. save ESP and SS in a CPU-internal register (but only if target segment selector's PL < CPL).
5. load SS and ESP from TSS ("")
6. push user SS ("")
7. push user ESP ("")
8. push user EFLAGS
9. push user CS
10. push user EIP
11. clear some EFLAGS bits
12. set CS and EIP from IDT descriptor's segment selector and offset

One of this bit is the mode bit.

Clear it → kernel mode.

x86 when an interrupt is fired (cont.)

1. decide the vector number (depends on the source of the interrupt)
2. fetch the interrupt descriptor from the IDT.
3. check that CPL <= DPL in the descriptor (but only if INT instruction).
4. save ESP and SS in a CPU-internal register (but only if target segment selector's PL < CPL).
5. load SS and ESP from TSS ("")
6. push user SS ("")
7. push user ESP ("")
8. push user EFLAGS
9. push user CS
10. push user EIP
11. clear some EFLAGS bits
12. set CS and EIP from IDT descriptor's segment selector and offset

Jump to the interrupt handler.

Handling Clock Interrupt

- The clock interrupt handler invokes the kernel’s thread scheduler.
- The scheduler then
  - Select the next thread to run.
  - Dispatch the control to that thread.

A Toy Implementation

By Pramook Khungurn

Processor

- We have a similar 32-bit processor as that in Lecture 7.
  - Each register is 32-bit.
  - 32-bit address space.
- Registers
  - R0, R1, R2, R3
  - SP (stack pointer)
  - IP (instruction pointer)
  - PMAR (page map address register)

Processor (cont.)

- PMAR is the similar to that in Lecture 7
  - Least significant bit is the user/kernel mode bit.
    - 0 -> kernel
    - 1 -> user
  - Next to least significant bit is interrupt enable bit.
    - 0 -> processor will not check for interrupt
    - 1 -> otherwise
  - When PMAR is 0, there’s no address translation.

Processor (cont.)

- And interrupt can be fired two ways:
  - Hardware
  - Software --- through INT instruction
- When an interrupt is fired:
  1. IP, R0, R1, R2, R3 is pushed on the stack, respectively.
  2. PMAR's last two bit is cleared.
  3. The CPU inspects the interrupt number, and jumps to the address specified in the interrupt vector.

Processor (cont.)

- Note that when an interrupt is fired:
  - Only the last two bits of PMAR is changed, so we don’t switch address space.
  - SP does not change.

Address Space Organization

- Each address space is a byte array of 2^{32} bytes.
- We organize the address space so that the first 2^{31} bytes of every address space belongs to the kernel. (Remember Problem 3 from the midterm?)
- This way, there's no need to worry about changing PMAR if the IP points somewhere in the kernel portion of the address space.
  - Changing PMAR does not effect the next instruction being executed at all.

Information about a thread to keep?

- Very similar to Lecture 8:
  - Thread state: UNUSED, RUNNABLE, WAITING
  - Pointer to its stack.
  - Its stack pointer
  - PMAR
- Other registers are kept in the stack.

Information about a thread (cont.)

```
struct threadentry {
    int state;
    int *stack;
    int sp;
    int pmar;
} threadtable[7];
```

Thread Scheduler

```
procedure RUNNEXT() {
    SCHEDULER();          // picks a new thread
    DISPATCH();           // switch to the thread
}
```

Thread Scheduler (cont.)

- There's a kernel variable "me" that contains the ID of the current thread.
- SCHEDULER() picks a new value of "me."
- Here, we use a simple round-robin scheduler.
- This is the same as that of Lecture 8.

Thread Scheduler (cont.)

```
procedure SCHEDULER() {
    me  = FIND_NEXT_RUNNABLE(me);
}

procedure FIND_NEXT_RUNNABLE(x) {
    do {
        x = (x + 1) % 7;
    } while (threadtable[x].state != RUNNABLE);
    return x;
}
```

Dispatcher

- Changes to another thread.
- What to do:
  - Loads the thread's PMAR.
  - Loads the stack pointer.
  - Pop R3, R2, R1, R0.
  - Return to the address on the stack.
- Everything has to be executed in the above order, why?

Dispatcher (cont.)

```
procedure DISPATCH() {
   PMAR = threadtable[me].pmar;
   SP = threadtable[me].sp;
   POP R3
   POP R2
   POP R1
   POP R0
}
```

Clock Interrupt Handler

- Things to do:
  - Save PMAR and SP.
  - Call RUNNEXT.

```
procedure CLOCK_INTERRUPT() {
   threadtable[me].pmar = PMAR | 3;
   threadtable[me].sp = SP;
   RUNNEXT();
}
```

Preemptive Scheduling of User Threads

Silberschatz

Section 4.2

One-to-one Model

- Don’t bother implement a thread manager in user address space. Just use the kernel thread.
- One user thread = one kernel thread.

One-to-one Model (cont.)

- Pros:
  - Easiest to implement (since there's nothing to write).
  - Every operating system supports this model.
    - Linux, Solaris 9, Windows 95, 98, 2000, and XP does not have built-in support for preemptive scheduling of user threads.

One-to-one Model (cont.)

- Cons:
  - Can be slow because of high overhead:
    - Thread creation. Very high if every kernel thread is a process.
    - Context switching

One-to-many Model

- One kernel thread (usually a process) corresponds to a number of user threads.
- Implements a thread manager in the kernel thread.

One-to-many Model (cont.)

- How to do preemptive scheduling?
  - Initially, the thread manager requests the OS to schedule a clock interrupt some time in the future.
  - Once the clock interrupt occurs, the OS sends a message (or signal) to the thread manager.
  - The thread manager has a message handler that gets evoked everything it receives a message.
  - The message handler calls YIELD() to give control to other user thread.

One-to-many Model (cont.)

- Pros:
  - Less overhead incurred by thread creation and thread switching.
- Cons:
  - If a user thread issues a blocking system call, then the all the threads in the same kernel threads also blocks.

Many-to-many Model

- User threads are multiplexed among many kernel threads.
- Kernel threads that manage user threads together must share address space.
  - Each of them is not a process.
- Need OS support.
  - Old versions of Solaris.
  - IRIX, HP-UX, Tru64 Unix

Many-to-many Model (cont.)

- Pros:
  - Cheap overhead like one-to-many model.
  - Better CPU utilization than one-to-many model.
- Cons:
  - Complex!
  - User threads share fate.
  - Kernel threads also share fate.

Many-to-many Model (cont.)

- Note that modern operating systems don't implement this feature.
- Why? (This is my theory. Take it with a grain of salt.)
  - They delegate this functionality to thread libraries so as to reduce complexity of the kernel?
  - Hardware is fast enough that context switching hardly matters?

Interprocess Communication

Silberschatz

Section 3.4, 3.Project, and 4.4.3

Interprocess Communication

- In our case, it is “interthread communication.”
- Why?
  - Information sharing: for example, shared files
  - Computational speedup: allow threads to cooperate
- Two approaches:
  - Shared memory
  - Message passing

Shared Memory

- Threads communicate by reading/writing to/from memory locations that they share.
- Threads in the same address space can do this directly.
- Threads in different address spaces must request the OS to modify their page tables so that they share at least a page.
  - This is done by MAP(id, block, page) system call.

Shared Memory (cont.)

- Pros:
  - Fast
    - Threads communicate directly by LOAD and STORE instructions.
  - Flexible
    - User can implement any communication mechanism he wants.
- Cons:
  - Not Fault Tolerant
    - If threads share memory, they share fate.
  - Burden on User
    - User must implement communication mechanism by himself.

Message Passing

- Threads communicate by sending messages.
- Kernel provides a service message sending.
- Typically, messages are received and sent to **mailbox** or **ports.**
- Example system calls:
  – int SEND(int mailbox_id, message_t message)
    - Send message to a particular mailbox.
  – int RECEIVE(int mailbox_id, message_t *buffer)
    - Get a message from a mailbox.

Message Passing (cont.)

- Kernel should also provide the following system calls:
  - int CREATE_MAILBOX()
    - Returns the ID of the new mailbox.
  - void DELETE_MAILBOX(int mailbox_id)
    - Delete the mailbox with the given ID.

Message Passing (cont.)

- Sending and receiving messages may be **blocking** or **non-blocking.**
  - **Blocking send:** The sending thread waits until the message is received by the receiver.
  - **Nonblocking send:** The call finishes as soon as the mailbox gets the message or report that it cannot send the message.
  - **Blocking receive:** The receiver blocks until a message it available in the mailbox.
  - **Nonblocking receive:** The receiver gets a message of no message.

Message Passing (cont.)

- Pros:
  - Fault Tolerance
    - No memory sharing.
- Cons
  - Slow
    - Everything is done through the kernel.
  - Short messages only
    - Mailboxes have limited capacity.
  - Inflexible
    - Fixed communication mechanism (but is actually very general).

Signals

- A limited form of interprocess communication in Unix operating systems.
- Used to inform a process (UNIX has processes, not threads) that an event occurs.
  - An interrupt is fired.
  - The process child has terminated.
  - Some other process kills the process.

Signals (cont.)

- The signal itself is an integer constant.

SIGABRT - process aborted
SIGALRM - signal raised by alarm
SIGBUS - bus error: "access to undefined portion of memory object"
SIGCHLD - child process terminated, stopped (*or continued)
SIGCONT - continue if stopped
SIGFPE - floating point exception: "erroneous arithmetic operation"
SIGHUP - hangup
SIGILL - illegal instruction
SIGINT - interrupt
SIGKILL - kill
SIGPIPE - write to pipe with no one reading
SIGQUIT - quit
SIGSEGV - segmentation violation
SIGSTOP - stop executing temporarily
SIGTERM - termination
SIGTSTP - terminal stop signal

etc.

Signals (cont.)

- Signals are handled much like interrupts.
- When a signal is sent to a process, the process's execution is interrupted.
- A function called **signal handler** is then called.
- Once the signal handler finishes execution, the process resumes execution.

Signals (cont.)

- The kernel supplies some default signal handlers.
- Each user process can also specifies its own signal handler.
  - Which means it can ignore some signals.
- However, the kernel forbids a process to specify handlers for some signals:
  - SIGKILL
  - SIGSTOP

Signals (cont.)

```
#include <signal.h>
#include <unistd.h>
#include <stdio.h>

void handel_SIGINT() {
	printf("Caught SIGINT.");
}

int main(int argc, char *argv[])
{
	struct sigaction handler;
	handler.sa_handler = handle_SIGINT;
	sigaction(SIGINT, &handler, NULL);

	while(1);

	return 0;
}
```

Signal (cont.)

- Think: How would you implement preemptive scheduling with signals?

Scheduling Algorithms

Silberschatz

Section 5.2 and 5.3

Scheduling

- When the kernel takes control of the CPU, it has to decide which thread to run next.
- This process is called **scheduling.** (**short-term scheduling** in Silberschatz.)
- We have seen that there are two main types of scheduling:
  - Nonpreemptive: The scheduler gets to run only when a thread calls it.
  - Preemptive: External mechanism invokes the scheduler from time to time.

Scheduling (cont.)

- Scheduling can affect:
  - Performance of your system.
  - Happiness of users.

Scheduling Criteria

- How to measure "goodness" of your scheduling algorithm?
  - CPU Utilization: How much time is the CPU busy?
  - Throughput: Number of processes completed per time unit.
  - Turnaround time: How long it takes to execute a process.
  - Waiting time: How long a process waits to be run.
  - Response time: Time from submission of a request until its completion.

Scheduling Criteria (cont.)

- Typically, we want to:
  - Maximize CPU Utilization
  - Maximize throughput
  - Minimize turnaround time
  - Minimize waiting time
  - Minimize response time
- In interactive systems, it is desirable to minimize the **variance** of response time.
  - User prefers predictable interactions.

Scheduling Algorithm

- Can be abstracted as follows:
  - Thread that are in “runnable” state is placed inside a list of runnable threads
  - The scheduling algorithm picks one thread out of the list and dispatch the CPU to it.

Scheduling Algorithms (cont.)

- Some common algorithms:
  - First-Come First-Served
  - Shortest-Job-First
  - Priority Scheduling
  - Round-Robin Scheduling
  - Multilevel Queue Scheduling
  - Multilevel Feedback-Queue Scheduling

First-Come First-Served (FCFS)

- The list of runnable process is a **queue.**
- A process that enters the queue before gets to run before.
- There is no preemption. Thread gets to run until it releases the CPU.

FCFS (cont.)

- This algorithm is the simplest of it all, but there are a lot of drawbacks:
  - Threads share fate.
  - Average turnaround time is usually high.
    - Threads that takes a lot of time to run adds to the turnaround time of other threads.
  - **Convoy effect:** A compute-intensive (means using a lot of CPU and little I/O) thread can slow down other I/O intensive threads.
  - Cannot be used in time-sharing system.

Shortest-Job-First (SJF)

- Pick the thread that the scheduler thinks will release the CPU the soonest as the next thread to run.
- Can be either preemptive or nonpreemptive:
  - Nonpreemptive: Allow the current thread to release CPU before selecting the next thread.
  - Preemptive: Once a thread with a smaller time to release CPU enters the queue, dispatch to that thread immediately.

SJF (cont.)

- Pros:
  - Gives optimal average turnaround time.
- Cons:
  - Hard to estimate the time until threads release CPU. (Can use some approximation though.)
  - Threads with high time-to-release-CPU may not get to run at all. (This is the problem of **starvation**.)

Priority Scheduling

- Each thread is associated with a numerical priority.
- There's a separate queue for each value of priorities.
- The scheduler selects a thread from the queue with highest priority. Usually this selection uses FCFS algorithm.

Priority Scheduling (cont.)

- Pros:
  - Flexible. Can be tuned to a particular application.
- Cons
  - Starvation: Some low priority threads might not get to run at all if there's a constant influx of high priority threads.
- We can solve the starvation problem by increasing a thread's priority as it stays in the queue longer. This technique is called **aging.**

Round Robin Scheduling (RR)

- Preemptive scheduling where each thread is given a time quantum.
- The list of runnable threads is a queue.
- The scheduler picks the thread at the front of the queue to run. Two things can happen:
  - The thread terminates or waits for something, in which the scheduler just picks a new thread.
  - The thread exhausts its time quantum, it is put back at the end of the queue.

RR (cont.)

- Pros:
  - Fault Tolerance
    - A thread cannot hog CPU forever.
  - Fair
    - No starvation. Every thread gets some share of CPU.
- Cons:
  - Average waiting time is often long.
  - Hard to determine the right time quantum to use.

Multilevel Queue Scheduling

- Have more than one queues of runnable threads.
- Each queue has:
  - Its own scheduling algorithm.
  - Its associated priority.
- Each thread is assigned permanently to one queue.

Multilevel Queue Scheduling (cont.)

- Two possibility of scheduling threads in different queues:
  - When a new thread is added to the queue with higher priority, the current running thread might be preempted if it belongs to the queue with lower priority.
  - Each queue has its own time quantum.

Multilevel Feedback-Queue Scheduling

- Have multiple queues like Multilevel Queue Scheduling.
- However, threads can be moved among queues.
  - If a thread uses too much CPU time, it is moved to lower priority queues.
  - If a thread stays in a low priority queue for too long time, it might be moved to a higher priority queue. (aging)

Multilevel Feedback-Queue Scheduling (cont.)

- Pros:
  - Very general
- Cons:
  - Very complex
  - Hard to select values for all the parameters
    - Algorithm for each queue
    - When to demote threads
    - When to promote threads

Multiprocessor Scheduling

Silberschatz

Section 5.4

Multiple CPUs

- **Load sharing** and **parallel processing** becomes possible.

Approaches

- Asymmetric multiprocessing
- Symmetric multiprocessing

Asymmetric Multiprocessing

- A CPU runs the kernel. Other CPU runs user threads.
- Client-Server architecture.
- Simple: Only one processor modifies the kernel’s data structure.
- But if the system has heavy load, then the kernel is the bottleneck.

Symmetric Multiprocessing

- All processors run both the kernel and user threads.
- Each processor schedules threads to run by itself.
- No bottleneck. Greater degree of parallelism.
- But the kernel must be programmed carefully.
  - Many processors may modify the kernel's data structure at the same time.
  - Need to ensure that the data contained therein are consistent.
- Most operating systems support this approach.

Issues

- Processor Affinity
- Load Balancing
- Symmetric Multithreading

Processor Affinity

- A processor has cache.
- When a thread runs on a processor, the cache of that processor is filled with data in the thread's address space.
- If you move a thread from one processor to another, the cache of the receiving processor needs to be cleared and repopulated with the data the thread accesses.
- This incurs a lot of overhead.
- So, the kernel should avoid moving a thread from one processor to another.

Load Balancing

- Load balancing is the act of trying to keep loads evenly distributed among processors.
- Necessary when each processor has its own scheduling queue.
- Load balancing is done by moving threads from one processor’s queue to another processor’s queue.

Load Balancing (cont.)

- Two kinds of load balancing:
  - **Push migration:** The kernel periodically checks the load on each processor, and redistributes the threads to even the loads.
  - **Pull migration:** An idle processor "steal" threads from a buy processor's queue.
- Load balancing conflicts with processor affinity. But every operating system needs both. Trade-off ensues.

Symmetric Multithreading

- Some CPU such as Intel with **hyperthreading** provides more than one **logical processors** from one **physical processor.**
- OS can think of logical processors as mutiple physical processors. → No need to change code.
- However, being aware of logical processors may help improve performance.
  - Don’t schedule threads from different address space on logical processors on the same physical processor.