Threads

Shamelessly taken from

6.033 Lecture Note Chapter 5.B & C

Virtual Memory

- Harden modularity by disallowing modules to read/write memory of other modules.
- Not a complete solution for modularity by itself.
  - Only memory abstraction.
- Still need to deal with:
  - Virtual interpreter
  - Virtual communication channel

Virtual Processor

- Virtual interpreter abstraction.
- Problem: One processor, but many modules.
- Goal: Give each module a **virtual processor**, that it can think of as its own.
  - Programmer doesn’t have to think about other programs that are running.
  - If a module screw ups, it only affects its own virtual processor. Not other modules.

Threads

- A **thread** is a module in execution.
- A thread is an abstraction that has enough information about the state of a module so that you can **stop** it, and later **resume** it.
- From the thread’s point of view, it continues doing its job, unaware of how many times it has been stopped or resumed.

Virtual Processors with Threads

- Associating one thread with one module.
- The module now has the illusion that it has the processor to itself.
- To share processors among modules, we have to switch between modules.
- This is simple. We just stop a module’s thread, and resume another module’s thread.

What info to include in a thread?

- The program counter.
- Values of the registers.
  - Stack Point (SP)
  - PMAR
  - Other registers used for calculations.
- Other information is available in the module’s virtual memory. So, we just ignore it.

When to switch thread?

- Most modules spend most of their time waiting for some conditions to be true.
- For example, an editor can be waiting for keyboard inputs.
- While waiting, a module does not do any useful work.
- When a module starts waiting, switch the thread so that other modules can use the processor.

Example

- **Editor thread**

```
WAIT_LOOP:
    if (input_count <= processed count)
        goto WAIT_LOOP;
```

- **Keyboard manager thread**

```
input_count++;
```

Example (cont.)

- The loop the editor is doing is called a **spin loop**.
  - It repeatedly checks a condition until the condition is true.
- input_count is the number of characters read by the keyboard.
- processed_count is the number of characters the editor has processed.

Example (cont.)

- When the keyboard manager receives a character from the keyboard, it increases input_count by one.
- The editor thread checks if there is a new character to process by checking if input_count > processed_count.
- Once the editor thread *consumes* a character, it increases processed_count by one.

Example (cont.)

- The editor thread should release its control of the processor once it enters WAIT_LOOP.
- WAIT_LOOP:

```
while (input_count <= processed_count)
        YIELD();
```

YIELD System Call

- Enters a part of the kernel called the **thread manager**.
- The thread manager chooses a thread to give the processor to, and changes the thread.

YIELD System Call (cont.)

```
procedure YIELD() {
    save this thread's state;
    schedule another thread to run;
    dispatch processor to that thread;
}
```

YIELD System Call (cont.)

- So, in the thread that is going to be scheduled, where does the execution resumes?
- At anywhere the state was saved:
  - The next instruction after the YIELD() system call in the thread that used YIELD().
  - Anywhere for any other threads.

Layering Thread Managers

- There can be multiple layers of thread managers.
- The processor is a thread manager with two threads.
  - Main thread used for computation.
  - Interrupt thread for handling interrupt.
  - Switching thread is done when an interrupt is fired. State savings is done by hardware.

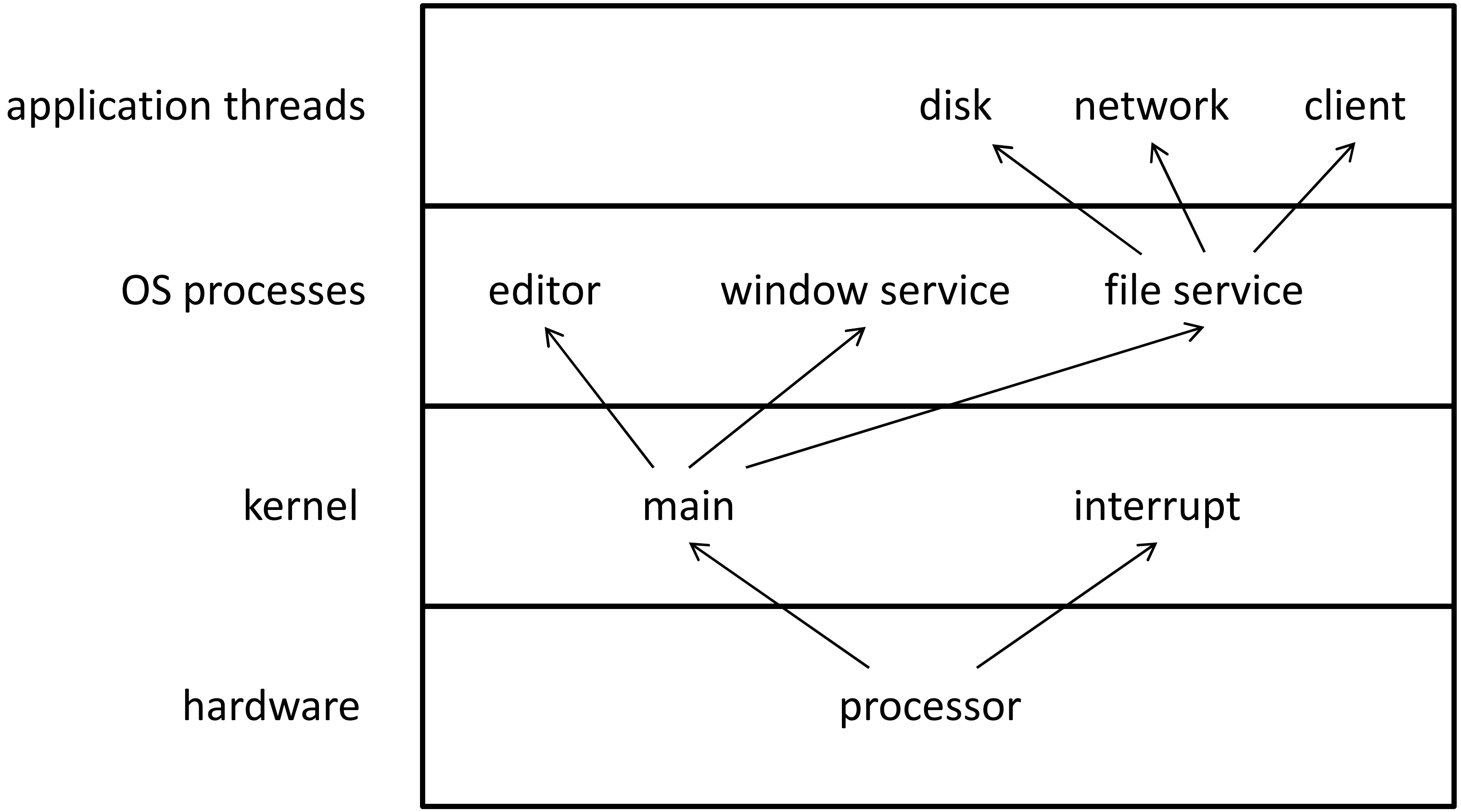
Layering Thread Managers (cont.)

- The main thread then contains another thread manager = the kernel.
  - Threads = operating system processes.
  - Allow processes to share the processors by periodically switching between them.
  - Use timer interrupt to signal thread switching.
  - We already talked about this. ☺

Layering Thread Managers (cont.)

- Each OS process may implement its own thread manager.
- And so on…

Layering Thread Managers (cont.)



Threads and Address Spaces

- Threads and address spaces are independent.
- Two or more threads can share an address space.
  - The kernel address space is shared by two threads.
    - The main kernel thread.
    - The interrupt thread.
  - Some user modules might have multiple threads using one shared address space.
  - If two threads have the same PMAR, then they use the samse address space.

Threads and Address Spaces (cont.)

- A thread may use more than one address space.
  - The main thread of the processor switches between multiple address spaces.

Process

- **Process** = a thread that owns its address space.
- A process can implement a thread manager, and can have multiple threads inside it.
- Most of the time, a process has only a single thread.
  - Such processes are simple, and so are common.

Process (cont.)

- A process may implement multiple threads to increase efficiency:
  - One thread may be busy waiting for input.
  - Other threads may compute.
- Implications of multiple threads in a process.
  - Don’t have to worry about switching address space. Every threads share the same address space.
  - But threads share fate. If one thread screws up, the other may as well be gone together.

Implementing a Thread Manager

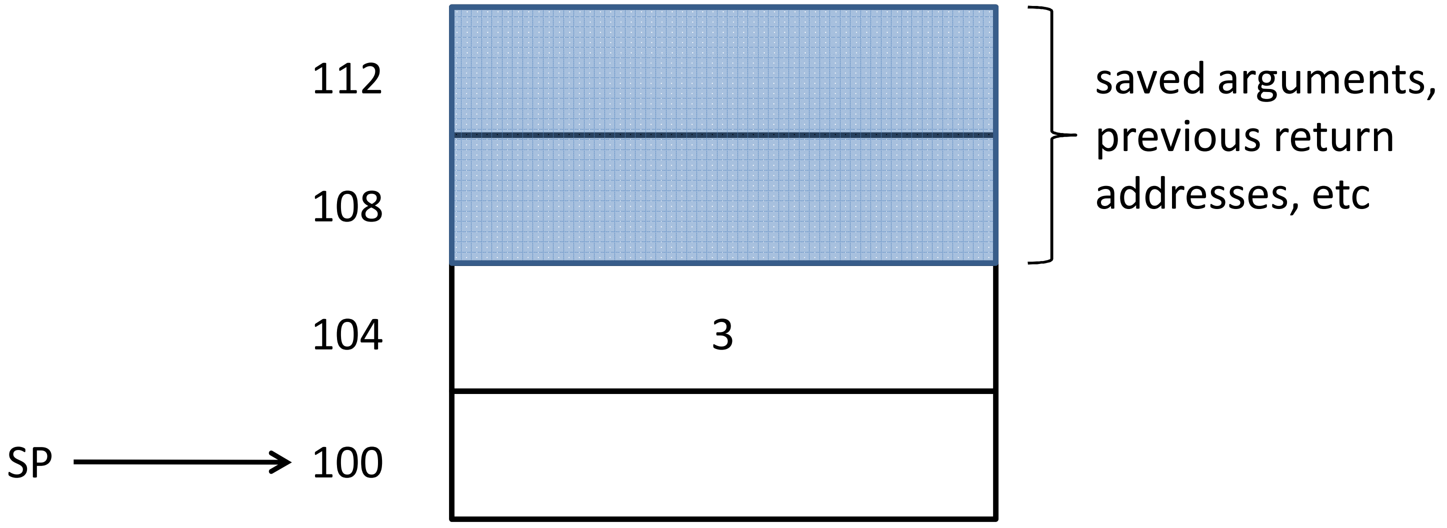
Switching Threads

- Let’s return to the busy waiting loop.

```
1   If (input_count <= processed_count) JMP 4
2   YIELD()
3   JMP 1
4   …
```

Switching Threads (cont.)

- When the above code calls YIELD(), the stack looks like this:



Switching Threads (cont.)

- We need to:
  - Save the stack pointer of the current thread.
  - Select a new thread to run.
  - Load the stack pointer of the new thread, and resume.
- So, let’s say we have an array:

  int threadtable[7];

  that store the stack pointers.
- A global variable “me” that olds the index of the current thread.

Switching Threads (cont.)

- Then, YIELD() may be implemented like this:

```
procedure YIELD() {
    threadtable[me] = SP;
    me = (me + 1) % 7;
    SP = threadtable[me];
}
```

Managing Threads

- We still need some way to:
  - Create new threads
  - Destroy threads
    - Once they have finished running
    - When some other threads requested them to be destroyed.
  - Manage variable number of threads.

Managing Threads (cont.)

- New calls of thread managers.
  - CREATE_THREAD(address)
    - address = where the thread must start execution.
  - EXIT_THREAD()
    - When a thread calls this function, it is terminated, cleanly.
  - DESTROY_THREAD(id)
    - Destroy the thread identified by id.

Managing Threads (cont.)

- threadtable needs some enhancement
  - Whether an entry is used or not.
  - The pointer to the chunk of memory holding the stack.
- Let’s assume for the moment that we allocate each thread an area of 4096 byte as a stack.

Managing Threads (cont.)

```
struct threadentry {
    bool used;
    int *stack;
    int stacktop;
} threadtable[7];
```

Managing Threads

- What does CREATE_THREAD needs to do?
  - Allocate the new stack.
  - Place the address of EXIT_THREAD on the stack.
  - Place the address given as argument on the stack.

Managing Threads (cont.)

```
procedure CREATE_THREAD(address) {
    k = FIND_UNUSED_ENTRY(threadtable)
    threadtable[k].used = true;
    threadtable[k].stack = ALLOC(4096);
    threadtable[k].stack[1023] = EXIT_THREAD;
    threadtable[k].stack[1022] = address;
    threadtable[k].stacktop = stack + 1021;
}
```

Managing Threads (cont.)

- YIELD also needs to be changed, slightly.

```
procedure YIELD() {
    threadtable[me].stacktop = SP;
    me = FIND_NEXT_USED_ENTRY(me);
    SP = threadtable[me].stacktop;
}
```

Managing Threads (cont.)

- FIND_NEXT_USED_ENTRY(me) returns the next entry after me in threadtable that is used. That is, threadtable[k].used = true.

```
procedure FIND_NEXT_USED_ENTRY(x) {
    do {
        x = (x + 1) % 7;
    } while (threadtable[x].used = false);
    return x;
}
```

Managing Threads (cont.)

- EXIT_THREAD() have to
  - Deallocate the stack of the current thread.
  - Free the threadtable cell by setting its status to "unused."
  - Find the next thread to run.

Managing Threads (cont.)

```
procedure EXIT_THREAD() {
    threadtable[me].used = false;
    DEALLOC(threadtable[me].stack);
    me = FIND_NEXT_USED_ENTRY(me);
    SP = threadtable[me].stacktop;
}
```

Managing Threads (cont.)

- DESTROY_THREAD is pretty much the same as EXIT_THREAD.
- Though we need to check whether the current thread wants to destroy itself or not.

Managing Threads (cont.)

```
procedure DESTROY_THREAD(id) {
    if (id == me)
        EXIT_THREAD();
    else {
        threadtable[id].used = false;
        DEALLOC(threadtable[id].stack);
    }
}
```

Sequence Coordination

- **Polling** = when a thread repeatedly checks a condition until it becomes true.
- Normally, it checks for a value of a shared variable.
- Polling is bad because the time a thread uses to poll something can be given to other threads that do computation.
- We want our thread manager to schedule threads so that those that do computation get the time it needs.

Sequence Coordination (cont.)

- Here’s what we do:
  - Have each thread tell the thread manager that it is waiting for something to be true.
  - Once the thread declares that, it is put in “WAITING” state, and its execution is suspended.
  - Other threads that update something that affects the condition can “notify” the thread manager. The thread manager can then check which other threads it can “wake up.”

Sequence Coordination (cont.)

- For illustration purpose, we’ll use the following two primitives:
  - WAIT(eventcount)
    - When a thread calls this, it tells the thread manager that it is waiting for the event that eventcount changes.
  - NOTIFY(eventcount)
    - When a thread calls this, it tells the thread manager that the value of eventcount has changed.

Sequence Coordination (cont.)

- Note that these primitives are just some way of achieving sequence coordination.
- Real systems have difference primitives.
  - In Linux, a process can wait() for another process to change state.
  - Java threads has wait() and notify() as well, but not as specific as ours.
  - We’re not dealing with semaphores, locks, and things like that yet. We’re not talking about sharing resources here.

Sequence Coordination (cont.)

- With the primitives in place, the editor’s busy wait loop can become:

Editor thread

```
while (*input_count <= processed_count)
    WAIT(input_count);
```

Sequence Coordination (cont.)

- The keyboard thread becomes:

Keyboard manager thread

```
    (*input_count)++;
    NOTIFY(input_count);
```

Sequence Coordination (cont.)

- With waiting, a thread can have three states.
  - WAITING = it is waiting for something.
  - RUNNABLE = the thread manager can schedule it to run, but is not running now.
  - RUNNING = it is currenly running.

- However, when implementing the thread manager, there’s no need to distinguish between RUNNABLE and RUNNING.

Sequence Coordination (cont.)

- Augmenting threadtable (again)
  - A threadtable entry may have one of the three states: UNUSED, WAITING, RUNNABLE.
  - Store the pointer to the eventcount that the thread is waiting.

```
struct threadentry {
    int state;
    int *stack;
    int *eventcount;
    int stacktop;
} threadtable[7];
```

Sequence Coordination (cont.)

- We’ll change YIELD() so that we **separate scheduling mechanism (how to switch to a new thread) from scheduling policy (how to select a new thread).**

```
procedure YIELD() {
    threadtable[me].stacktop = SP;
    RUNNEXT();
}
```

Sequence Coordination (cont.)

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

Sequence Coordination (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;
}
```

Sequence Coordination (cont.)

```
procedure DISPATCH() {
    SP = threadtable[me].stacktop;
}
```

Sequence Coordination (cont.)

- WAIT
  - Sets a thread’s state to WAITING.
  - Tests the eventcount again and reset the thread’s state if the test succeeds.
  - Call the scheduler.

Sequence Coordination (cont.)

```
procedure WAIT(eventcount, value) {
    threadtable[me].eventcount = eventcount;
    threadtable[me].state = WAITING;
    RUNNEXT();
}
```

Sequence Coordination (cont.)

- NOFITY loops over all threads, and wake up threads that have the eventcount as the given one.

Sequence Coordination (cont.)

```
procedure NOTIFY(eventcount) {
  for(i=0;i<7;i++) {
      if (threadtable[i].state == WAITING) &&
       (threadtable[i].eventcount == eventcount)
           threadtable[i].state = RUNNABLE;
  }
}
```

Things to be careful about…

- The code we have seen so far only works when there is one processor.
  - We have only one “me,” but each processor has to have its own “me.”
  - Two processors cannot run the same thread at the same time. So, we have to make sure that they have different “me.”
  - What if a thread call DESTROY_THREAD, while the thread being destroyed is being run by another processor?

Things to be careful about… (cont.)

- Shoving the “me” issue aside, we still have more problems with WAIT and NOTIFY in multi-processor setting.
- An event may be lost if one thread calls NOTIFY while the other thread is calling WAIT.
- See an example next page.
- We’ll deal with synchronization and all that jazz after the midterm.

Things to be careful about… (cont.)

Thread 0 (in WAIT)

```
threadtable[me].eventcount =
      eventcount;

threadtable[me].state = WAITING;
RUNNEXT();
```

Thread 1 (in NOTIFY)

```
if (threadtable[0].status  == WAITING) &&
   (threadtable[0].eventcount == eventcount)
      threadtable[0].state = RUNNABLE;

if (threadtable[1].status  == WAITING) &&
   (threadtable[1].eventcount == eventcount)
      threadtable[1].state = RUNNABLE;
```

Types Scheduling

- So far, thread switching only happens when a thread calls YIELD or WAIT.
- This is called **nonpreemptive scheduling**: a thread releases a processor when it wants to.
- Nonpreemptive scheduling is bad because if a thread does not release a processor, then all other threads will not have a chance to run.

Types of Scheduling (cont.)

- Some systems use **cooperative multitasking**: it requires each thread to call YIELD from time to time.
- This is not good because it is only a convention.
- If a thread does not call YIELD then other threads are screwed.

Types of Scheduling (cont.)

- To enforce modularity, we need **preemptive scheduling**: the thread manager forces threads to give up the processor after it has run for a while, say 100 milliseconds.
- In this way, a thread that does not give up the processor cannot stop other threads from progressing.

Preemptive Scheduling

- Implementing preemptive scheduling is quite complex.
- There needs to be an external mechanism that signals the thread manager to do its job.
  - Typically, a clock device firing interrupt every 100ms is used.

Preemptive Scheduling (cont.)

- The interrupt can occur anywhere, not at predefined location like the call to YIELD() in nonpreemptive scheduling.
- Therefore, we need to store more states per thread:
  - The values of EVERY register. (Not just the stack pointer as in nonpreemptive scheduling.)
  - The instruction pointer (so that we can return later.)
- Saving states and things like this must be done with the help of hardware.

Preemptive Scheduling (cont.)

- Things get more complicated because only the kernel can handle interrupts.
- How can we implement preemptive scheduling in user processes?
  - Well, with some help from the OS, of course.
  - More on this after the midterm.