CSE 153
Design of Operating Systems
Summer 2022

Lecture 5/6: Threads and Concurrency
Processes

- Recall that …
  - A process includes:
    » An address space (defining all the code and data pages)
    » OS resources (e.g., open files) and accounting info
    » Execution state (PC, SP, regs, etc.)
    » PCB to keep track of everything
  - Processes are completely isolated from each other

- But…
Some issues with processes

- Creating a new process is costly because of new address space and data structures that must be allocated and initialized
  - Recall struct proc in xv6 or Solaris

- Communicating between processes is costly because most communication goes through the OS
  - Inter Process Communication (IPC) – we will discuss later
  - Overhead of system calls and copying data
Rethinking Processes

- What is similar in these cooperating processes?
  - They all share the same code and data (address space)
  - They all share the same privileges
  - They all share the same resources (files, sockets, etc.)

- What don’t they share?
  - Each has its own execution state: PC, SP, and registers

- Key idea: Separate resources from execution state

- Exec state also called thread of control, or thread
Recap: Process Components

- A process is named using its process ID (PID)
- A process contains all of the state for a program in execution

<table>
<thead>
<tr>
<th>Per-Process State</th>
</tr>
</thead>
<tbody>
<tr>
<td>- An address space</td>
</tr>
<tr>
<td>- The code for the executing program</td>
</tr>
<tr>
<td>- The data for the executing program</td>
</tr>
<tr>
<td>- A set of operating system resources</td>
</tr>
<tr>
<td>» Open files, network connections, etc.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Per-Thread State</th>
</tr>
</thead>
<tbody>
<tr>
<td>- An execution stack encapsulating the state of procedure calls</td>
</tr>
<tr>
<td>- The program counter (PC) indicating the next instruction</td>
</tr>
<tr>
<td>- A set of general-purpose registers with current values</td>
</tr>
<tr>
<td>- Current execution state (Ready/Running/Waiting)</td>
</tr>
</tbody>
</table>
Threads

- Separate execution and resource container roles
  - The thread defines a sequential execution stream within a process (PC, SP, registers)
  - The process defines the address space, resources, and general process attributes (everything but threads)

- Threads become the unit of scheduling
  - Processes are now the containers in which threads execute
  - Processes become static, threads are the dynamic entities
Threads

- Separate execution and resource container roles
  - The thread defines a sequential execution stream within a process (PC, SP, registers)
  - The process defines the address space, resources, and general process attributes (everything but threads)

- Threads become the unit of scheduling
  - Processes are now the containers in which threads execute
  - Processes become static, threads are the dynamic entities
Recap: Process Address Space

- **Code (Text Segment)**
- **Static Data (Data Segment)**
- **Heap (Dynamic Memory Alloc)**
- **Stack**

Address Space:
- **0x00000000**
- **0xFFFFFFFF**

Control Flow:
- **PC**
- **SP**
Threads in a Process

- Stack (T1)
- Stack (T2)
- Stack (T3)
- Heap
- Static Data
- Code

Threads:
- Thread 1
- Thread 2
- Thread 3

Process Control (PC):
- PC (T1)
- PC (T2)
- PC (T3)
Thread Design Space

<table>
<thead>
<tr>
<th>Address Space</th>
<th>Thread</th>
</tr>
</thead>
<tbody>
<tr>
<td>One Thread/Process</td>
<td>Many Threads/Process</td>
</tr>
<tr>
<td>One Address Space (MSDOS)</td>
<td>One Address Space (Pilot, Java)</td>
</tr>
<tr>
<td>Many Address Spaces (Early Unix)</td>
<td></td>
</tr>
<tr>
<td>Many Address Spaces (Mac OS, Unix, Windows)</td>
<td></td>
</tr>
</tbody>
</table>
Process/Thread Separation

- Separating threads and processes makes it easier to support multithreaded applications
  - Concurrency does not require creating new processes

- Concurrency (multithreading) can be very useful
  - Improving program structure
  - Handling concurrent events (e.g., Web requests)
  - Writing parallel programs

- So multithreading is even useful on a uniprocessor
Using fork() to create new processes to handle requests in parallel is overkill for such a simple task.

Recall our forking Web server:

```c
while (1) {
    int sock = accept();
    if ((child_pid = fork()) == 0) {
        Handle client request
        Close socket and exit
    } else {
        Close socket
    }
}
```
Threads: Concurrent Servers

- Instead, we can create a new thread for each request

```c
web_server() {
    while (1) {
        int sock = accept();
        thread_fork(handle_request, sock);
    }
}

handle_request(int sock) {
    Process request
    close(sock);
}
```
Implementing threads

- **Kernel Level Threads**
  - All thread operations are implemented in the kernel
  - The OS schedules all of the threads in the system
  - Don’t have to separate from processes

- OS-managed threads are called **kernel-level threads** or **lightweight processes**
  - Windows: **threads**
  - Solaris: **lightweight processes (LWP)**
  - POSIX Threads (pthreads): **PTHREAD_SCOPE_SYSTEM**
Sample Thread Interface

- `thread_fork(procedure_t)`
  - Create a new thread of control
  - Also `thread_create()`, `thread_setstate()`
- `thread_stop()`
  - Stop the calling thread; also `thread_block`
- `thread_start(thread_t)`
  - Start the given thread
- `thread_yield()`
  - Voluntarily give up the processor
- `thread_exit()`
  - Terminate the calling thread; also `thread_destroy`
Thread Scheduling

- The thread scheduler determines when a thread runs
- It uses queues to keep track of what threads are doing
  - Just like the OS and processes
  - But it is implemented at user-level in a library
- Run queue: Threads currently running (usually one)
- Ready queue: Threads ready to run
- Are there wait queues?
  - How would you implement thread_sleep(time)?
Non-Preemptive Scheduling

- Threads voluntarily give up the CPU with `thread_yield`

```c
while (1) {
    printf("ping\n");
    thread_yield();
}
```

```c
while (1) {
    printf("pong\n");
    thread_yield();
}
```

- What is the output of running these two threads?
The semantics of \texttt{thread\_yield} are that it gives up the CPU to another thread.

- In other words, it \texttt{context switches} to another thread.

So what does it mean for \texttt{thread\_yield} to return?

Execution trace of ping/pong:
- \texttt{printf(“ping\n”);}
- \texttt{thread\_yield();}
- \texttt{printf(“pong\n”);}
- \texttt{thread\_yield();}
- …
Threads Summary

- Processes are too heavyweight for multiprocessing
  - Time and space overhead
- Solution is to separate threads from processes
  - Kernel-level threads much better, but still significant overhead
  - User-level threads even better, but not well integrated with OS
- Scheduling of threads can be either preemptive or non-preemptive

- Now, how do we get our threads to correctly cooperate with each other?
  - Synchronization…
Synchronization

- For correctness, we need to control this cooperation
  - Threads *interleave executions arbitrarily* and at different rates
  - Scheduling is not under program control

- We control cooperation using *synchronization*
  - Synchronization enables us to restrict the possible inter-leavings of thread executions
What about processes?

- Does this apply to processes too?
  - Yes!

- Processes are a little easier because they don’t share by default

- But share the OS structures and machine resources so we need to synchronize them too
  - Basically, the OS is a multi-threaded program
Shared Resources

We initially focus on coordinating access to shared resources

- **Basic problem**
  - If two concurrent threads are accessing a shared variable, and that variable is read/modified/written by those threads, then access to the variable must be controlled to avoid erroneous behavior

- **Over the next couple of lectures, we will look at**
  - Exactly what problems occur
  - How to build mechanisms to control access to shared resources
    » Locks, mutexes, semaphores, monitors, condition variables, etc.
  - Patterns for coordinating accesses to shared resources
    » Bounded buffer, producer-consumer, etc.
A First Example

- Suppose we have to implement a function to handle withdrawals from a bank account:

  ```
  withdraw (account, amount) {
      balance = get_balance(account);
      balance = balance – amount;
      put_balance(account, balance);
      return balance;
  }
  ```

- Now suppose that you and your father share a bank account with a balance of $1000.

- Then you each go to separate ATM machines and simultaneously withdraw $100 from the account.
We’ll represent the situation by creating a separate thread for each person to do the withdrawals. These threads run on the same bank machine:

```c
withdraw (account, amount) {
    balance = get_balance(account);
    balance = balance - amount;
    put_balance(account, balance);
    return balance;
}
```

What’s the problem with this implementation?
- Think about potential schedules of these two threads.
Interleaved Schedules

The problem is that the execution of the two threads can be interleaved:

What is the balance of the account now?
Shared Resources

- Problem: two threads accessed a shared resource
  - Known as a race condition (remember this buzzword!)

- Need mechanisms to control this access
  - So we can reason about how the program will operate

- Our example was updating a shared bank account

- Also necessary for synchronizing access to any shared data structure
  - Buffers, queues, lists, hash tables, etc.
When Are Resources Shared?

- **Local variables?**
  - Not shared: refer to data on the stack
  - Each thread has its own stack
  - Never pass/share/store a pointer to a local variable on the stack for thread T1 to another thread T2

- **Global variables and static objects?**
  - Shared: in static data segment, accessible by all threads

- **Dynamic objects and other heap objects?**
  - Shared: Allocated from heap with malloc/free or new/delete
How Interleaved Can It Get?

How contorted can the interleavings be?

- We'll assume that the only atomic operations are reads and writes of individual memory locations.
  - Some architectures don't even give you that!
- We'll assume that a context switch can occur at any time.
- We'll assume that you can delay a thread as long as you like as long as it's not delayed forever.

```plaintext
get_balance(account);
balance = get_balance(account);
balance = ...................................
balance = balance – amount;
balance = balance – amount;
put_balance(account, balance);
put_balance(account, balance);
```
What do we do about it?

- Does this problem matter in practice?
- Are there other concurrency problems?
- And, if so, how do we solve it?
  - Really difficult because behavior can be different every time
- How do we handle concurrency in real life?
Mutual Exclusion

- Mutual exclusion to synchronize access to shared resources
  - This allows us to have larger atomic blocks
  - What does atomic mean?

- Code that uses mutual called a critical section
  - Only one thread at a time can execute in the critical section
  - All other threads are forced to wait on entry
  - When a thread leaves a critical section, another can enter
  - Example: sharing an ATM with others

- What requirements would you place on a critical section?
Critical Section Requirements

Critical sections have the following requirements:

1) Mutual exclusion (mutex)
   - If one thread is in the critical section, then no other is

2) Progress
   - A thread in the critical section will eventually leave the critical section
   - If some thread T is not in the critical section, then T cannot prevent some other thread S from entering the critical section

3) Bounded waiting (no starvation)
   - If some thread T is waiting on the critical section, then T will eventually enter the critical section

4) Performance
   - The overhead of entering and exiting the critical section is small with respect to the work being done within it
About Requirements

There are three kinds of requirements that we'll use:

- **Safety** property: nothing bad happens
  - Mutex
- **Liveness** property: something good happens
  - Progress, Bounded Waiting
- **Performance** requirement
  - Performance

Properties hold for each run, while performance depends on all the runs.
- Rule of thumb: When designing a concurrent algorithm, worry about safety first (but don't forget liveness!).
Mechanisms For Building Critical Sections

- Locks
  - Primitive, minimal semantics, used to build others

- Semaphores
  - Basic, easy to get the hang of, but hard to program with

- Monitors
  - High-level, requires language support, operations implicit

- Architecture help
  - Atomic read/write
    » Can it be done?
How do we implement a lock?

First try

```c
pthread_trylock(mutex) {
    if (mutex==0) {
        mutex= 1;
        return 1;
    } else return 0;
}
```

- Does this work? Assume reads/writes are atomic
- The lock itself is a critical region!
  - Chicken and egg
- Computer scientist struggled with how to create software locks

Thread 0, 1, ...

...//time to access critical region
while(!pthread_trylock(mutex); // wait
<critical region>
pthread_unlock(mutex)
This is called **alternation**

It **satisfies mutex:**

- If blue is in the critical section, then turn == 1 and if yellow is in the critical section then turn == 2
- (turn == 1) ≡ (turn != 2)

Is there anything wrong with this solution?
Third try – two variables

We added two variables to try to break the race for the same variable

Is there anything wrong with this solution?
Fourth try – set before you check

flag[0] = 1;
while (flag[1] != 0);
critical section
flag[0]=0;
outside of critical section

flag[1] = 1;
while (flag[0] != 0);
critical section
flag[1]=0;
outside of critical section

Is there anything wrong with this solution?
Fifth try – double check and back off

```c
flag[0] = 1;
while (flag[1] != 0) {
    flag[0] = 0;
    wait a short time;
    flag[0] = 1;
}
critical section
flag[0]=0;
outside of critical section

flag[1] = 1;
while (flag[0] != 0) {
    flag[1] = 0;
    wait a short time;
    flag[1] = 1;
}
critical section
flag[1]=0;
outside of critical section
```
Six try – Dekker’s Algorithm

Bool flag[2];
Int turn = 1;

flag[0] = 1;
while (flag[1] != 0) {
    if (turn == 2) {
        flag[0] = 0;
        while (turn == 2);
        flag[0] = 1;
    } //if
} //while

critical section
flag[0] = 0;
turn = 2;
outside of critical section

flag[1] = 1;
while (flag[0] != 0) {
    if (turn == 1) {
        flag[1] = 0;
        while (turn == 1);
        flag[1] = 1;
    } //if
} //while

critical section
flag[1] = 0;
turn = 1;
outside of critical section
Another solution: Peterson's Algorithm

```c
int turn = 1;
bool try1 = false, try2 = false;

while (true) {
    try1 = true;
    turn = 2;
    while (try2 && turn != 1) ;
    critical section
    try1 = false;
    outside of critical section
}

while (true) {
    try2 = true;
    turn = 1;
    while (try1 && turn != 2) ;
    critical section
    try2 = false;
    outside of critical section
}
```

• This satisfies all the requirements
• Here's why...
Mutex with Atomic R/W: Peterson's Algorithm

while (true) {
    { ¬ try1 ∧ (turn == 1 ∨ turn == 2) }
1 try1 = true;
    { try1 ∧ (turn == 1 ∨ turn == 2) }
2 turn = 2;
    { try1 ∧ (turn == 1 ∨ turn == 2) }
3 while (try2 && turn != 1) ;
        { try1 ∧ (turn == 1 ∨ ¬ try2 ∨
                  (try2 ∧ (yellow at 6 or at 7)) )
                  critical section
3 critical section
        }
4 try1 = false;
    { ¬ try1 ∧ (turn == 1 ∨ turn == 2) }
outside of critical section
}

(blue at 4) ∧ try1 ∧ (turn == 1 ∨ ¬ try2 ∨ (try2 ∧ (yellow at 6 or at 7))
 ∧ (yellow at 8) ∧ try2 ∧ (turn == 2 ∨ ¬ try1 ∨ (try1 ∧ (blue at 2 or at 3)))
... ⇒ (turn == 1 ∧ turn == 2)

while (true) {
    { ¬ try2 ∧ (turn == 1 ∨ turn == 2) }
5 try2 = true;
    { try2 ∧ (turn == 1 ∨ turn == 2) }
6 turn = 1;
    { try2 ∧ (turn == 1 ∨ turn == 2) }
7 while (try1 && turn != 2) ;
        { try2 ∧ (turn == 2 ∨ ¬ try1 ∨
                  (try1 ∧ (blue at 2 or at 3)) )
                  critical section
7 critical section
        }
8 try2 = false;
    { ¬ try2 ∧ (turn == 1 ∨ turn == 2) }
outside of critical section
}
Some observations

- This stuff (software locks) is hard
  - Hard to get right
  - Hard to prove right

- It also is inefficient
  - A spin lock – waiting by checking the condition repeatedly

- Even better, software locks don’t really work
  - Compiler and hardware reorder memory references from different threads
    - Something called memory consistency model
    - Well beyond the scope of this class 😊

- So, we need to find a different way
  - Hardware help; more in a second