Operating Systems (234123)  
*threads vs. processes*

[~ chapter #3 in Feitelson’s OS notes]

Dan Tsafrir (Sunday, 14/5/2017)
What’s a process – reminder

- **Running instance of a program**
  - Has: data, stack, code, PC pointing to code, registers
  - Possible to run several instances of same program
- **Associated with a PCB**
  - “Process control block”
  - Saves all kernel objects & info associated with the process
    - PID, open files, priority...
  - Save state of process when not running (registers)
- **Associated with a state**
  - Running
  - Ready (to run)
  - Waiting
Multiprocessing

• **Definition**
  – Using several cores (or processors) for running a single job

• **Motivation**
  – Finish the job faster
  – Provide better service (increased responsiveness) to users

• **Example**
  – Web server, which gets requests and returns replies
  – Replies can be static (existing files) or dynamic (computed on the fly)
Web server implementation

- **Problem – without multiprocessing**
  - while ( true )
    - `<client, request> = get_next_request();`
    - `reply = process(request); /* wait for it to finish... */`
    - `send(client, reply);`
  - Simple, but processing can take a long time (potentially doing slow I/O), delaying other pending requests

- **Solution – with multiprocessing**
  - while ( true ):
    - `<client, request> = get_next_request();`
    - `if( fork() == 0 ) /* multi-process! child is doing the work*/`
      - `reply = process(request);`
      - `send(client, reply);`
  - Not as simple, but processing is done in parallel on different cores
  - Comment: often using bounded number of pre-fork(ed) children (why?)
How about matrix multiplication?

- Let A and B be matrices and assume we want to compute $C = A \times B$
  - The matrices are quite big
  - But the physical memory is big enough to hold them

- **Question**
  - Assuming we have a 4-way multicore, can we use it to speed up the computation in a similar manner to the web server?

- **Answer**
  - Sure. For example, let’s divide C into quarters and have each core run its own process and compute its own quarter
How about matrix multiplication?

- **Problem**
  - The OS protects the 4 processes from one another
  - They are isolated and don’t share
  - Each has its own memory space and, specifically, its own data & heap

- **So we must pay a cost**
  - Need 4 copies of matrices
  - Worse, all copies might be bigger than the memory
  - Also, need to perform lots of communication (=copying) such that one core will be able to combine the four quarters into one result (details: soon)
Solution: multithreading

- Computing entities must have their own registers and stack
- But all of their other state (including data & heap) could be shared
Solution: multithreading

- Multithreading = a process has/contains several “threads of execution”, sharing everything but the stack & registers
Solution: multithreading

- (Sometimes people use the following illustration, which depicts exactly the same thing)
Standard API for threads

- **OpenMP**
  - Stands for “Open Multi-Processing”
  - Implemented in C, C++, and Fortran
  - In gcc, compile with flag `-fopenmp`
  - Consists of a set of compiler ‘pragma’ directives, such that the code looks like it’s serial, but compiler directives make it parallel:
    ```
    #pragma omp parallel for
    for (i = 0; i < N; i++)
      arr[i] = 2 * i;
    ``
  - Namely, the serial and parallel versions of the code are unified (it’s the same code)
  - Supported by all major operating systems (including Windows)
  - More:
Standard API for threads

- Often implemented natively by language runtimes
  - Java (no fork)
  - C++ (as of 2011 – std::thread)
  - ...
Standard API for threads

• pthreads
  – Recall: POSIX is the standard for UNIX operating systems
    • POSIX = “portable operating system interface”
  – pthreads Stands for “POSIX threads”
  – A standard C library
    • Include <pthread.h> and when compiling use gcc –lpthread ...
  – Also implemented on Windows
    • pthreads-w32, which know also support 64bit...
  – The library we use in this course
    • We’ll soon see an example
## Sharing

<table>
<thead>
<tr>
<th></th>
<th>unique to process</th>
<th>unique to pthread</th>
</tr>
</thead>
<tbody>
<tr>
<td>registers (notably PC)</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>execution stack</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>memory address space</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>open files</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>per-open-file position</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>working directory</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>user/group credentials</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>signal handling</td>
<td>Y</td>
<td>N</td>
</tr>
</tbody>
</table>

...  

- **In some OSes, the kernel implements processes and threads differently**  
  - Such that internally in the kernel each process really is a container of threads  
  - Examples: Windows and Solaris
## Sharing

<table>
<thead>
<tr>
<th></th>
<th>unique to process</th>
<th>unique to pthread</th>
</tr>
</thead>
<tbody>
<tr>
<td>registers (notably PC)</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>execution stack</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>memory address space</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>open files</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>per-open-file position</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>working directory</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>user/group credentials</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>signal handling</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **In Linux, internally, ‘process’ and ‘thread’ are exactly the same**
  - Process = thread = “task”; each has its own PCB (= “task_struct”)
  - Logically, every item in above table is a pointer in task_struct
  - Logically, when 2 tasks share an item, it means they both point to it
- **Users have fine-grain control on what’s being shared via the ‘clone’ syscall**
  - fork() and pthread_create() are simply wrapper functions of clone()
Linux’s clone system call

• Creating a thread
  – clone( child_stack=0x420cc260, flags=CLONE_VM|CLONE_FS|
    CLONE_FILES|CLONE_SIGHAND|CLONE_THREAD|
    CLONE_SYSVSEM|CLONE_SETTLS|CLONE_PARENT_SETTID|
    CLONE_CHILD_CLEARTID, parent_tidptr=0x420cc9e0, 
    tls=0x420cc950, child_tidptr=0x420cc9e0 )

• Creating a process
  – clone( child_stack=0, flags=CLONE_CHILD_CLEARTID |
    CLONE_CHILD_SETTID|SIGCHLD, 
    child_tidptr=0x7f4936ecc770 )

• See ‘man clone’
  – No need to understand every flag, just the principle
  – Example: if CLONE_VM is set, then parent+child share memory space
Intermediate summary

• So far
  – Abstract discussion

• Next
  – Give an example:
    1. For communication via explicit “message passing”
       – Which is required when using processes, because they don’t share
    2. For communication via “shared memory”
       – Which is possible for threads, because they share an address space
In POSIX, “everything is a file”

• “File descriptor”
  – A number that represents an I/O “channel” on some device

• It can be obtained, for example, like so
  – int fd = open("/some/file", flags);

• Given a file descriptor, one can interact with the underlying device via the ‘read’ and ‘write’ system calls
  – int read(int fd, char *buf, size_t n);
  – int write(int fd, char *buf, size_t n);

• Another way to obtain a file descriptor is....
Message passing communication – example

• We will discuss a (POSIX) communication channel called a “pipe”
  – There are other types of communication channels that allow us to send messages, but this is perhaps the simplest example
  – We’ll use the pipe to deliver a single message from one process (parent) to another (child)

• A pipe is a pair of two “file descriptors” (aka “fd” for short)
  – int pipe_fd[2]
  – Each integer is a handle to a kernel communication object, such that
    • pipe_fd[0] = read side of the channel
    • pipe_fd[1] = write side of the channel
    • Everything written via pipe_fd[1] can be read via pipe_fd[0]
    • Writing and reading is actually done through the kernel, using the write() and read() system calls
    • The kernel saves everything that is being written via pipe_fd[1], such that later it’ll be able to serve subsequent reads from pipe_fd[0]
Message passing communication – example

• **Code:**
  – int pipe_fd[2];
  – char src_buf[N] = “...”;
  – char dst_buf[N];
  – `write( pipe_fd[1], src_buf, N );`
    • Go to the kernel and copy N bytes from src_buf to the communication channel identified by pipe_fd[1]
  – `read( pipe_fd[0], dst_buf, N );`
    • Go to the kernel and copy to dst_buf the N bytes previously written to the communication channel identified by pipe_fd[1], which, as noted, is associated with pipe_fd[0]

• **Note**
  – When a running process attempts to read through pipe_fd[0] but nothing has been written yet via pipe_fd[1], the process would block
/*
 * DO_SYS: a safe way to invoke system calls:
 */

#define DO_SYS( syscall ) do {
  /* safely invoke a system call */
  if( (syscall) == -1 ) {
    perror( #syscall );
    exit(1);
  }
} while( 0 )

/* why do we need the do-while? why not just the if? */
Message passing communication – example

char g_msg[N];    /* “g” stands for “global” */
enum {RD=0, WT=1};
int main() { // send g_msg from parent to child
    int fd[2];
    DO_SYS( pipe(fd) ); // establish communication channel
    if( fork() != 0 ) { // parent writes, so
        DO_SYS( close(fd[RD]) ); // close read side
        fill_g_msg();        // don’t care how
        DO_SYS( write(fd[WT], g_msg, N ) );
        DO_SYS( wait(NULL ) ); // for child to end
    }
    else {            // child reads, so
        DO_SYS( close(fd[WT]) ); // close write side
        DO_SYS( read(fd[RD], g_msg, N ) );
    }
    return 0;
}
Message passing communication – example

- **Actually, there are two g_msg arrays**
  - One for parent and another for child
  - They appear to be the same array, but they are not
  - Reason: with fork, everything is a copy

- **So actually, there are 4 file descriptors**
  - 2 fds for parent, and 2 fds for child
  - They appear to be the same fd-pair, but they are not (though both read-s are connected to both write-s)
  - Reason: with fork, everything is a copy

- **The write / read system calls**
  - Copy g_msg to and from the kernel

- **Question:**
  - Can child send msg back to parent?

- **A file descriptor is in fact**
  - An index to a kernel array of channels
// Assume that filling g_msg requires a lot of 
// computational work.  
// So we want to use 2 threads, one thread to fill the 
// first half of g_msg and another to fill the second half
void fill_g_msg( void )
{
    pthread_t t1, t2;

    // launch the two threads
    pthread_create(&t1, NULL, thread_fill, "first");
    pthread_create(&t2, NULL, thread_fill, "second");

    // wait for both threads to finish
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
}
**Shared memory communication – example**

```c
void* thread_fill(void *arg)
{
    // initially, assume I’m the “first” thread
    int i;
    int lo = 0;
    int hi = N/2;

    if( strcmp((char*)arg, “second”) == 0 ) {
        // I’m the “second” thread, filling 2nd half
        lo = N/2 + 1;
        hi = N - 1;
    }

    for(i = lo; i <= hi; i++)
        g_msg[i] = /*do some really hard work here*/ ;

    return null;
}
```
void* thread_fill(void *arg) {
    // initially, assume I'm the "first" thread
    int i;
    int lo = 0;
    int hi = N/2;

    if( strcmp((char*)arg, "second") == 0) {
        // I'm the "second" thread
        lo = N/2 + 1;
        hi = N - 1;
    }

    for(i = lo; i <= hi;
        g_msg[i] = /*do some really hard work here*/;
    }
    return null;
}
Bottom line

• **Thread or process?**
  – Depends on how you want your tasks to communicate

• **Threads = communication is done via shared memory**
  – In principle, more efficient, as doesn’t create a copy (“zero copy”)
  – But often means we have to access the memory in a synchronized manner, which, as we will see, could be *challenging* to get right
    • Right = correct & efficient (more on that in future lectures)
  – When synchronization is not an issue (because it’s done very coarsely, or because we have a programming model / data structure that hides synchronization issues from ordinary programmers), then it is easy and convenient

• **Processes = communication is done via explicit message passing**
  – Tasks explicitly exchange messages via read/write or send/receive
  – Sometimes requires making a copy of data => less efficient
  – Typically requires the use of system calls => less efficient
  – Much easier to understand, get right, and reason about (prove stuff)
Let’s discuss

THE COST OF CONTEXT SWITCHING
Cost of context switch is inherently relative

- The overall overhead of context switching is largely determined by how long tasks run before they are preempted
  - Assume that a context switch takes $C$ time (say, microseconds)
  - Assume tasks run at least $K \cdot C$ microseconds before they are preempted
  - Then the rate of cycles we spend on context switching is:
    - $\frac{C}{(K \cdot C + C)} = \frac{1}{K+1}$ // $K > 0$, real number (not necessarily integral)
    - Namely, we spend $\frac{1}{K+1}$ fraction of the time on context switching
    - E.g., with $K=1$, half the time is spent on context switching
  - Consequently, bigger $K$ implies a smaller cost
  - Or, in other words, the longer tasks run before they are preempted, the smaller the cost of context switching becomes

- How is $K$ determined?
  - By the kernel (quantum duration)
  - By the application (how long can it run before having to wait)
Context switch overhead consists of 2 components

- **Direct overhead**
  - How long it takes for the kernel to save the state (= registers) of the previous task and resume the state of the next task
  - Can be roughly measured within the kernel as follows
    - `cycles_t before = get_cycles(); /* CPU cycles; nanosecs if 1GHz */`
    - `do_context_switch(prev_task, next_task);`
    - `cycles_t after = get_cycles();`
    - `cycles_t direct_overhead = after - before;`

- **Indirect overhead**
  - The time it takes for the hardware to reconstruct the state it (= the hardware) created in order to accelerate the execution of the task
  - The hardware does this reconstruction while the task is running
  - To understand this component, we need to have some idea about how processors work
Computer architecture ("MAMAS") in a nutshell

CACHING & THE MEMORY HIERARCHY
Problem: CPU faster than main memory

• Reminder
  – milli=1:1000, micro=1:1,000,000, nano=1:1,000,000,000

• CPU
  – Each core is able to do up to billions of operations in one second
    • If the speed of the processors is “1GHz”, it (roughly) means that the processor can do one billion ($10^9$) instructions per second
    • Or one instruction per nanosecond
  – The time-per-instruction is called the “cycle” of the CPU

• Main memory (DRAM)
  – Has a latency (= time it takes to read/write data from/to memory) which is typically longer than 100 CPU cycles
  – So memory is nowadays oftentimes >100x slower than CPU
Problem: CPU faster than main memory

Not so long ago, this problem was called “the memory wall”...

Not so long ago, this problem was called “the memory wall”...

- CPU: 60% per year (2x in 1.5 years)
- DRAM: 9% per year (2x in 10 years)

Gap grew 50% per year (exponentially)
Problem: CPU faster than main memory

More recently: CPUs aren’t getting that much faster, but memory bandwidth might become an issue if more cores simultaneously access the memory...
Empirical observation

• **Principle of Locality (= Locality of Reference)**
  – A phenomenon commonly observed while computer programs run:
  – The collection of the memory locations that are referenced in a short period of time by a running program often consists of relatively well predictable, small clusters of locations

• **Two important special cases:**
  – Temporal locality
    • If at one point in time a particular memory location is referenced, then it is likely that the same location will be referenced again in the near future
  – Spatial locality
    • If a particular memory location is referenced at a particular time, then it is likely that nearby memory locations will be referenced in the near future
    • (With these definitions, temporal locality is said to be a special case of spatial locality)
Locality of reference: example

Mystery sorting function #3

Time

Address

OS (234123) – threads vs. processes 35
Locality of reference: example
Solution: caching

• Cache is
  – Smaller-but-faster hardware memory structure (faster than main memory)
  – Used to hold a few recently used DRAM locations
  – The act of holding said recently used locations (in the cache structure is) called “caching”

• How it works
  – Handled by hardware, automatically
  – That is, the cache is being filled by the hardware, on the fly
  – While the CPU generates memory accesses (read and write ops), the hardware **MMU** (memory management unit) arranges things such that the accessed locations are being cached
  – When cache space runs out, the hardware must also choose which locations to evict from the cache

• Why it works
  – Because of the principle of locality
Cache hierarchy: Intel i7-4770 (Haswell, Q2’13)

• 4 cores; core speed
  – 3.4 GHz (Turbo Boost off)
  – 3.9 GHz (max on)

• Memory hierarchy
  – 32 KB L1d + 32 KB L1i = 64 KB L1 cache (per core)
    • Latency = 4–5 cycles ≈ 1.2–1.5 nanoseconds
  – 256 KB L2 (per core)
    • Latency = 12 cycles ≈ 3.5 nanoseconds
  – 8 MB L3 (shared)
    • Latency = 36 cycles ≈ 11 nanoseconds
  – Up to 32 GB DRAM (shared)
    • Latency = 230 cycles ≈ 68 nanoseconds

OS (234123) – threads vs. processes
LET’S RETURN TO THE COST OF CONTEXT SWITCH
The indirect context switch component

• **Includes**
  – The time it takes for to (re)populate the DRAM caches with useful content after the context switch occurs – aka “warn up the caches”
  – There are other such HW components

• **Threads vs. processes**
  – If threads actually do work on same data / utilize same instructions (not always the case), then the indirect overhead of context switching could be smaller
  – So switching between threads could be cheaper
  – (This is why the 2.4 Linux scheduler gave a bonus to tasks sharing the same address space; see previous lecture)

• **Direct vs. indirect**
  – Experimental data shows that indirect component of context switching might reach as high as two orders of magnitude more than the direct component (depending on what’s running)
FYI-S
Processes can share memory too

• Via system calls
  1. `shmget(key, sizem, attributes)` // get
  2. `shmat(key, address, attributes)` // attach
  3. `shmdt(address)` // detach
  4. `shmctl(key, command, struct shmid_ds *buf)` // control
  5. `shm_open` // as file
  6. `shm_unlink` // as file

• Different (virtual) memory addresses that refer to the same (physical) memory location

• Homework
  – Try it: share memory between two processes
The copy-on-write optimization

• The fork() system call creates a copy of the address space of the parent
  – But it only creates a logical copy
  – There is no physical copy until we really need to have one
  – Which is when either child or parent write

• Even then, copying is not of the entire address space
  – OS copies only the “page” of the target memory location (typically 4KB)
  – One page at a time
  – More on that in lectures to follow
Terminology

- **Multitasking**
  - Having multiple processes time slice on the same core

- **Multiprogramming**
  - Having multiple jobs in the system (either on the same core or on different cores)

- **Multiprocessing**
  - Using multiple processors for the same job or system in parallel
USER-LEVEL THREADS
Threads can be implemented in user-level

• Motivation
  – Sometimes programmers have domain-specific knowledge that allows them to implement multithreading in a more efficient manner
  • Creating a stack for a function in user-level can be done in only a few 10s of cycles (as opposed to kernel-level, which will usually take at least 1000s of cycles)
  – Typically (not always), this is done for “runtimes” that present to users some programming model that makes it easier for them to exploit parallelism
  – Typically (not always), such runtimes allow tasks to run to completion once they start to run
Example – OmpSs [https://pm.bsc.es/ompss]

```c
#pragma omp task inout( arr[lo:hi] )
void sort ( int lo, int hi, int *arr ) {
    if ( hi-lo < THRESHOLD )
        sequential_sort ( lo, hi, arr );
    else { // in parallel, while tracking dependencies...
        int mid = (lo+hi) / 2;
        sort( lo, mid, arr ); // this task
        sort ( mid+1, hi, arr); // is done in parallel with this task
        merge (lo,mid,hi, arr); // while this task waits
    }
}
```

• **A function can be declared a task**
  – Which means it is run in parallel with other instances of that function so long as there are no input/output dependencies

• **The OmpSs runtime automatically tracks tasks’ input/output dependencies**
  – It runs a task only after its inputs are ready
  – E.g., ‘merge’ (also a task) must wait for the two ‘sort’s that come before it
Example – OmpSs [https://pm.bsc.es/ompss]

• **The OmpSs runtime maintains one OS-thread per core**
  – On each core it does its own scheduling of OmpSs “tasks”
  – The kernel is not aware of this at all

• **Which means tasks are not allowed to make system calls**
  – That is, they are allowed, but then the core will stand idle (as far as the OmpSs runtime is concerned), because the kernel is not aware of the fact that there are other waiting tasks
  – So users are instructed not to invoke system calls in OmpSs tasks
setjmp & longjmp

- **Standard C library functions**
  - Which allow programmers to do their own user-level scheduling
  - Again, OS remains unaware (any blocking syscall will block all of them)
- **The gist of it**
  - `switch() {
    if( setjmp( g_buf[g_current] ) == 0 )
        schedule();   // run another thread...
    else
        // and we’re back...
  }
  `schedule() {
    new = select_next_thread_to_run();
    g_current = new;
    longjmp( g_buf[g_current] ); // back to corresponding else
  }

- **Homework:** read page 32 in Feitelson’s OS notes
- **Also** [https://www.cs.purdue.edu/homes/cs240/lectures/Lecture-19.pdf](https://www.cs.purdue.edu/homes/cs240/lectures/Lecture-19.pdf)