CS 471 Operating Systems

Yue Cheng

George Mason University
Fall 2017
Review: Process
Process launching

Loading:
Takes on-disk program and reads it into the address space of process
Process executing

Diagram showing the relationship between CPU, Memory, Disk, and the process execution. The PC (Program Counter) points to the code in memory, which is divided into code and static data sections. The stack is also shown within the process memory. The diagram explains the loading process, which takes an on-disk program and reads it into the address space of the process.
Process State Transitions

- **Ready**
- **Running**
- **Blocked**

**Event occurs**

- From **Ready** to **Running** via **Dispatched**
- From **Running** to **Ready** via **Timeout**

**Event wait**

- From **Running** to **Blocked**
- From **Blocked** to **Running**
Process State Transitions

- **Ready**
  - Event occurs: E.g., I/O done

- **Blocked**
  - Event wait: E.g., I/O initiate

- **Running**
  - Dispatched
  - Timeout

- Transition arrows indicate:
  - From Ready to Running: Dispatched
  - From Running to Ready: Timeout
  - From Blocked to Ready: Event occurs
  - From Blocked to Running: Event wait
Outline

- Process communication
- Thread
- Multithreaded application
- Thread implementation
Process Communication
Process Communication

- Mechanism for processes to communicate and to synchronize their actions.

- Two models
  - Communication through a shared memory region
  - Communication through message passing
Previously, in a distributed system, message-passing was the only possible communication model. However, remote direct memory access (RDMA) technique bridges this gap by providing remote memory access through network.
Communication through Message Passing

- Message system – processes communicate with each other *without* resorting to *shared variables*

- A message-passing facility must provide at least two operations:
  - `send(message, recipient)`
  - `receive(message, recipient)`

- With *indirect communication*, the messages are sent to and received from *mailboxes* (or, *ports*)
  - `send (A, message) /* A is a mailbox */`
  - `receive (A, message)`
Communication through Message Passing

- Message passing can be either *blocking* *(synchronous)* or *non-blocking* *(asynchronous)*
  - **Blocking Send**: The sending process is blocked until the message is received by the receiving process or by the mailbox
  - **Non-blocking Send**: The sending process resumes the operation as soon as the message is received by the kernel
  - **Blocking Receive**: The receiver blocks until the message is available
  - **Non-blocking Receive**: “Receive” operation does not block; it either returns a valid message or a default value (null) to indicate a non-existing message
Communication through Shared Memory

- The memory region to be shared must be explicitly defined
- Using system calls – in Linux:
  - `shmget` creates a shared memory block
  - `shmat` maps an existing shared memory block into a process’s address space
  - `shmdt` removes (“unmaps”) a shared memory block from the process’s address space
  - `shmctl` is a general-purpose function allowing various operations on the shared block (receive information about the block, set the permissions, lock in memory, …)
- Problems with simultaneous access to the shared variables
- Compilers for concurrent programming languages can provide direct support when declaring variables (e.g. “shared int buffer”)

Communication through Shared Memory
Shared Memory Example

```c
#include <stdio.h>
#include <sys/shm.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

int main()
{

    pid_t pid;
    /* the identifier for the shared memory segment */
    int segment_id;
    /* a pointer to the shared memory segment */
    char *shared_memory;
    /* the size (in bytes) of the shared memory segment */
    const int segment_size = 4096;

    /** allocate a shared memory segment */
    segment_id = shmget(IPC_PRIVATE, segment_size, S_IRUSR | S_IWUSR);

    /** attach the shared memory segment */
    shared_memory = (char *) shmat(segment_id, NULL, 0);
    printf("shared memory segment %d attached at address %p\n", segment_id, shared_memory);

    /** write a message to the shared memory segment */
    sprintf(shared_memory, "Hi there CS 471 \n");

    return 0;
}
```
/* Do a fork */
pid = fork();

if (pid == 0){
    printf("\nIn the execl child, PID is %d memory is %p\n", pid, shared_memory);
    execvp("./sharechild.exe", "sharechild.exe", shared_memory, NULL);
    return 0;
}
else if (pid > 0) {
    wait(NULL);
    printf("In the parent, done waiting\n");
}

/** now print out the string from shared memory */
printf("\nIn Parent==>%s\n", shared_memory);

/** now detach the shared memory segment */
if ( shmdt(shared_memory) == -1) {
    fprintf(stderr, "Unable to detach\n");
}

/** now remove the shared memory segment */
shmctl(segment_id, IPC_RMID, NULL);

return 0;
#include <stdio.h>
#include <sys/shm.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

int main(int argc, char **argv)
{

    /**Print out the string from shared memory */
    printf("\nFrom Child==>%s\n\n", argv[1]);

    return 0;
}
Output

./share.exe
shared memory segment 720896 attached at address 0x10b74c000

In the execl child, PID is 0 memory is 0x10b74c000

From Child==>Hi there CS 471!
*

In the parent, done waiting

In Parent==>Hi there CS 471!
*
Thread
Thread

- A process, as defined so far, has only one thread of execution

- **Idea:** Allow multiple threads of execution within the same process environment, to a large degree independent of each other
Process vs. Thread

- Multiple threads within a process will share
  - The address space
  - Open files
  - Other resources

- Why thread?
  - Great potential for efficient and close cooperation
Single- vs. Multi-threaded Process
Multithreading

**Per Process Items**
Address Space
Global Variables
Open Files
Accounting Information

**Per Thread Items**
Program Counter
Registers
Stack
State
Single- vs. Multi-threaded Process

Single-Threaded Process Model

- Process Control Block
- User Stack
- Kernel Stack
- User Address Space

Multithreaded Process Model

- Thread Control Block
- User Stack
- Kernel Stack
- User Address Space

Single Threaded and Multithreaded Process Models
Each thread can be in any one of the several states, just like processes: **Ready, Running, Blocked**

Each thread has its own stack
Benefits

- **Resource Sharing**
  - Sharing the address space and other resources may result in high degree of cooperation
Benefits

- **Resource Sharing**
  - Sharing the address space and other resources may result in high degree of cooperation

- **Economy**
  - Creating/managing processes much more time consuming than managing threads: e.g., context switch
Benefits

- **Resource Sharing**
  - Sharing the address space and other resources may result in high degree of cooperation

- **Economy**
  - Creating/managing processes much more time consuming than managing threads: e.g., context switch

- **Better Utilization of Multicore Architectures**
Benefits

- **Resource Sharing**
  - Sharing the address space and other resources may result in high degree of cooperation

- **Economy**
  - Creating/managing processes much more time consuming than managing threads: e.g., context switch

- **Better Utilization of Multicore Architectures**

- **Responsiveness**
  - Multithreading an interactive application may allow a program to continue running even if part of it is blocked or performing a lengthy operation
Example Multithreaded Applications

- A multithreaded web server

![Diagram of a multithreaded web server]

- Web server process
- Dispatcher thread
- Worker thread
- Web page cache
- Network connection
- User space
- Kernel space
Example Multithreaded Applications

- A multithreaded web server

![Diagram of a multithreaded web server](image-url)

- Web server process
- Dispatcher thread
- Worker thread
- Web page cache
- Network connection
- User space
- Kernel space

Requests
while (TRUE) {
    get_next_request(&buf);
    handoff_work(&buf);
}

while (TRUE) {
    wait_for_work(&buf);
    check_cache(&buf; &page);
    if (not_in_cache)
        read_from_disk(&buf, &page);
    return_page(&page);
}
Example: Memcached

- Memcached—A high-performance memory-based caching system
  - 14k lines of C source code
  - https://memcached.org/

- A typical multithreaded server implementation
  - Pthread + libevent
  - A dispatcher thread dispatches newly coming connections to the worker threads in a round-robin manner
  - Event-driven: Each worker thread is responsible for serving requests from the established connections
Demo: Memcached
Multithreading vs. Multi-processes

- Real-world debate
  - Memcached vs. Redis
- Redis—A single-threaded memory-based data store
  - [https://redis.io/](https://redis.io/)
Wish List for Redis

- Explicit memory management.
- Deployable (Lua) Scripts. Talked about near the start.

- Multi-threading. Would make cluster management easier. Twitter has a lot of “tall boxes,” where a host has 100+ GB of memory and a lot of CPUs. To use the full capabilities of a server a lot of Redis instances need to be started on a physical machine. With multi-threading fewer instances would need to be started which is much easier to manage.

http://goo.gl/N9UTKD
Threads Execution

- When a multithreaded process is run on a single CPU system, the threads take turns to run.

```
| single core | T_1 | T_2 | T_3 | T_4 | T_1 | T_2 | T_3 | T_4 | T_1 | ...
|-------------|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----
| core 1      | T_1 | T_3 | T_1 | T_3 | T_1 | ... |
| core 2      | T_2 | T_4 | T_2 | T_4 | T_2 | ... |
```

time
Demo:
Memcached & Memaslap
Threads in Multicore Platforms

- **Challenge**: modify old programs and design new programs that are multithreaded

- **Issues**
  - Dividing activities
  - Balance
  - Data splitting
  - Data dependency
  - Testing and debugging
Thread Implementation
Implementing Threads

- Processes usually start with a single thread
- Usually, library procedures are invoked to manage threads
  - `Thread_create`: typically specifies the name of the procedure for the new thread to run
  - `Thread_exit`
  - `Thread_join`: blocks the calling thread until another (specific) thread has exited
  - `Thread_yield`: voluntarily gives up the CPU to let another thread run
- Threads may be implemented in
  - `User space`
  - `Kernel space`
User-level Threads

- User threads are supported above the kernel and are implemented by a thread library at the user level.
- The library (or runtime system) provides support for thread creation, scheduling and management with no support from the kernel.
User-level Threads (cont.)

- When threads are managed in user space, each process needs its own private **thread table** to keep track of the threads in that process
  - The thread table keeps track only of the per-thread items (program counter, stack pointer, register, state…)

- When a thread does something that **may** cause it to become blocked **locally** (e.g. wait for another thread), it calls a runtime system procedure
  - If the thread must be put into blocked state, the procedure performs **thread switching**
User-level Threads: Pros

- The operating system does not need to support multithreading
- Since the kernel is not involved, thread switching may be very fast
- Each process may have its own customized thread scheduling algorithm
- Thread scheduler may be implemented in the user space very efficiently
User-level Threads: Cons

- The implementation of *blocking system calls* is highly problematic (e.g. read from the keyboard)
  - *All the threads in the process risk being blocked!*

- Possible Solutions:
  - Change all system calls to non-blocking
  - Sometimes it may be possible to tell in advance if a call will block (e.g. *select* system call in some versions of Unix) \(\rightarrow\) “jacket code” around system calls
Kernel-level Threads

- Kernel threads are supported directly by the OS: The kernel performs thread creation, scheduling and management in the kernel space
Kernel-level Threads (cont.)

- The kernel has a thread table that keeps track of all threads in the system.
- All calls that *might* block a thread are implemented as system calls (greater cost).
- When a thread blocks, the kernel may choose another thread from the same process, or a thread from a different process.
Hybrid Implementations

- An alternative solution is to use kernel-level threads, and then multiplex user-level threads onto some or all of the kernel threads.
- A kernel-level thread has some set of user-level threads that take turns using it.
Pthread

- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization

- API specifies behavior of the thread library, implementation is up to development of the library

- Common in UNIX operating systems
## Pthread APIs

<table>
<thead>
<tr>
<th>Thread Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pthread_create</td>
<td>Create a new thread in the caller’s address space</td>
</tr>
<tr>
<td>pthread_exit</td>
<td>Terminate the calling thread</td>
</tr>
<tr>
<td>pthread_join</td>
<td>Wait for a thread to terminate</td>
</tr>
<tr>
<td>pthread_mutex_init</td>
<td>Create a new mutex</td>
</tr>
<tr>
<td>pthread_mutex_destroy</td>
<td>Destroy a mutex</td>
</tr>
<tr>
<td>pthread_mutex_lock</td>
<td>Lock a mutex</td>
</tr>
<tr>
<td>pthread_mutex_unlock</td>
<td>Unlock a mutex</td>
</tr>
<tr>
<td>pthread_cond_init</td>
<td>Create a condition variable</td>
</tr>
<tr>
<td>pthread_cond_destroy</td>
<td>Destroy a condition variable</td>
</tr>
<tr>
<td>pthread_cond_wait</td>
<td>Wait on a condition variable</td>
</tr>
<tr>
<td>pthread_cond_signal</td>
<td>Release one thread waiting on a condition variable</td>
</tr>
</tbody>
</table>
# Pthread APIs

<table>
<thead>
<tr>
<th>Thread Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pthread_create</td>
<td>Create a new thread in the caller’s address space</td>
</tr>
<tr>
<td>pthread_exit</td>
<td>Terminate the calling thread</td>
</tr>
<tr>
<td>pthread_join</td>
<td>Wait for a thread to terminate</td>
</tr>
<tr>
<td>pthread_mutex_init</td>
<td>Create a new mutex</td>
</tr>
<tr>
<td>pthread_mutex_destroy</td>
<td>Destroy a mutex</td>
</tr>
<tr>
<td>pthread_mutex_lock</td>
<td>Lock a mutex</td>
</tr>
<tr>
<td>pthread_mutex_unlock</td>
<td>Unlock a mutex</td>
</tr>
<tr>
<td>pthread_cond_init</td>
<td>Create a condition variable</td>
</tr>
<tr>
<td>pthread_cond_destroy</td>
<td>Destroy a condition variable</td>
</tr>
<tr>
<td>pthread_cond_wait</td>
<td>Wait on a condition variable</td>
</tr>
<tr>
<td>pthread_cond_signal</td>
<td>Release one thread waiting on a condition variable</td>
</tr>
</tbody>
</table>
Example of Using Pthread

```c
#include <stdio.h>
#include <assert.h>
#include <pthread.h>

void *mythread(void *arg) {
    printf("%s\n", (char *) arg);
    return NULL;
}

int main(int argc, char *argv[]) {
    pthread_t p1, p2;
    int rc;
    printf("main: begin\n");
    rc = pthread_create(&p1, NULL, mythread, "A"); assert(rc == 0);
    rc = pthread_create(&p2, NULL, mythread, "B"); assert(rc == 0);
    // join waits for the threads to finish
    rc = pthread_join(p1, NULL); assert(rc == 0);
    rc = pthread_join(p2, NULL); assert(rc == 0);
    printf("main: end\n");
    return 0;
}
```
HW 1 posted on Blackboard
Due 1:30pm 09/18 in class
Hard copy submission only
Windows XP Threads

- Windows XP supports kernel-level threads
- The primary data structures of a thread are:
  - ETHREAD (executive thread block)
    - Thread start address
    - Pointer to parent process
    - Pointer to the corresponding KTHREAD
  - KTHREAD (kernel thread block)
    - Scheduling and synchronization information
    - Kernel stack (used when the thread is running in kernel mode)
    - Pointer to TEB
  - TEB (thread environment block)
    - Thread identifier
    - User-mode stack
    - Thread-local storage
Linux Threads

- In addition to `fork()` system call, Linux provides the `clone()` system call, which may be used to create threads.

- Linux uses the term `task` (rather than process or thread) when referring to a flow of control.

- A set of flags, passed as arguments to the `clone()` system call determine how much sharing is involved (e.g. open files, memory space, etc.).