CS 333
Introduction to Operating Systems
Class 3 - Threads & Concurrency

Jonathan Walpole
Computer Science
Portland State University
Process creation in UNIX

- All processes have a unique process id
  - `getpid()`, `getppid()` system calls allow processes to get their information

- Process creation
  - `fork()` system call creates a copy of a process and returns in both processes (parent and child), but with a different return value
  - `exec()` replaces an address space with a new program

- Process termination, signaling
  - `signal()`, `kill()` system calls allow a process to be terminated or have specific signals sent to it
Example: process creation in UNIX

csh (pid = 22)

```c
...  
    pid = fork();
    if (pid == 0) {
      // child...
      ...
      exec();
    }
    else {
      // parent
      wait();
    }
...  
```
**Process creation in UNIX example**

**csh (pid = 22)**

```c
...  
    pid = fork();  
    if (pid == 0) {  
        // child...  
        ...  
            exec();  
    }  
    else {  
        // parent  
        wait();  
    }  
...  
```

**csh (pid = 24)**

```c
...  
    pid = fork();  
    if (pid == 0) {  
        // child...  
        ...  
            exec();  
    }  
    else {  
        // parent  
        wait();  
    }  
...  
```
Process creation in UNIX example

csh (pid = 22)

```
...  
pid = fork();  
if (pid == 0) {  
  // child...  
  ...  
  exec();  
}  
else {  
  // parent  
  wait();  
}  
...```

csh (pid = 24)

```
...  
pid = fork();  
if (pid == 0) {  
  // child...  
  ...  
  exec();  
}  
else {  
  // parent  
  wait();  
}  
...```
Process creation in UNIX example

csh (pid = 22)

...  
    pid = fork();
    if (pid == 0) {
        // child...
        ...
        exec();
    }
    else {
        // parent
        wait();
    }
...

csh (pid = 24)

...  
    pid = fork();
    if (pid == 0) {
        // child...
        ...
        exec();
    }
    else {
        // parent
        wait();
    }
...
Process creation in UNIX example

csh (pid = 22)

...  
pid = fork();  
if (pid == 0) {  
    // child...
    ...
    exec();
  }  
else {  
    // parent
    wait();
}  
...

ls (pid = 24)

//ls program
main(){  
    //look up dir  
    ...
}

Process creation (fork)

- Fork creates a new process by *copying* the calling process
- The new process has its own
  - memory address space
    - Instructions (copied from parent)
    - Data (copied from parent)
    - Stack ?? (empty)
  - register set (copied from parent)
  - Process table entry in the OS
### What other process state does the OS manage?

<table>
<thead>
<tr>
<th>Process management</th>
<th>Memory management</th>
<th>File management</th>
</tr>
</thead>
<tbody>
<tr>
<td>Registers</td>
<td>Pointer to text segment</td>
<td>Root directory</td>
</tr>
<tr>
<td>Program counter</td>
<td>Pointer to data segment</td>
<td>Working directory</td>
</tr>
<tr>
<td>Program status word</td>
<td>Pointer to stack segment</td>
<td>File descriptors</td>
</tr>
<tr>
<td>Stack pointer</td>
<td></td>
<td>User ID</td>
</tr>
<tr>
<td>Process state</td>
<td></td>
<td>Group ID</td>
</tr>
<tr>
<td>Priority</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scheduling parameters</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Process ID</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Parent process</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Process group</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Signals</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Time when process started</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU time used</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Children’s CPU time</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Time of next alarm</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Example fields of a process table entry**
Threads

- Processes have the following components:
  - an address space
  - a collection of operating system state
  - a CPU context ... or thread of control

- On multiprocessor systems, with several CPUs, it would make sense for a process to have several CPU contexts (threads of control)
  - Fork creates a new thread but not memory space
  - Multiple threads of control could run in the same memory space on a single CPU system too!
Threads

- Threads share a process address space with zero or more other threads

- Threads have their own
  - PC, SP, register state, stack

- A traditional process can be viewed as a memory address space with a single thread
Single thread state within a process

User Address Space

- **Stack**
  - `routine1` var1();
  - var2();

- **Text**
  - `main()`
  - `routine1()`
  - `routine2()`

- **Data**
  - `arrayA`
  - `arrayB`

- **Heap**

- **Stack Pointer**
  - `Prgm. Counter`
  - `Registers`

- **Process ID**
  - `Group ID`
  - `User ID`

- **Files**
  - `Locks`
  - `Sockets`
Multiple threads in an address space
What is a thread?

- A thread executes a stream of instructions
  - it is an abstraction for control-flow
- Practically, it is a processor context and stack
  - Allocated a CPU by a scheduler
  - Executes in the context of a memory address space
Summary of private per-thread state

Things that define the state of a particular flow of control in an executing program:

- Stack (local variables)
- Stack pointer
- Registers
- Scheduling properties (i.e., priority)
Shared state among threads

Things that relate to an instance of an executing program (that may have multiple threads)

- User ID, group ID, process ID
- Address space
  - Text
  - Data (off-stack global variables)
  - Heap (dynamic data)
- Open files, sockets, locks

Changes made to shared state by one thread will be visible to the others

- Reading and writing memory locations requires synchronization! ... a major topic for later ...
Independent execution of threads

Each thread has its own stack
How do you program using threads?

Split program into routines to execute in parallel

- True or pseudo (interleaved) parallelism

Alternative strategies for executing multiple routines
Why program using threads?

- Utilize multiple CPU’s concurrently
- Low cost communication via shared memory
- Overlap computation and blocking on a single CPU
  - Blocking due to I/O
  - Computation and communication
- Handle asynchronous events
Thread usage

A word processor with three threads
Processes versus threads - example

- A WWW process

GET / HTTP/1.0

HTTPD

disk
Processes versus threads - example

- A WWW process

Why is this not a good web server design?
Processes versus threads - example

- A WWW process

```
GET / HTTP/1.0
```

![Diagram showing processes versus threads](image)
Processes versus threads - example

- A WWW process

```
GET / HTTP/1.0
```

```
HTTPD
```

```
GET / HTTP/1.0
```

```
disk
```
Processes versus threads - example

- A WWW process

```
GET / HTTP/1.0
GET / HTTP/1.0
GET / HTTP/1.0
GET / HTTP/1.0
```
Threads in a web server

A multithreaded web server
Thread usage

- Rough outline of code for previous slide
  (a) Dispatcher thread
  (b) Worker thread

```c
while (TRUE) {
    get_next_request(&buf);
    handoff_work(&buf);
}

while (TRUE) {
    wait_for_work(&buf)
    look_for_page_in_cache(&buf, &page);
    if (page_not_in_cache(&page)
        read_page_from_disk(&buf, &page);
    return_page(&page);
}
```
# System structuring options

<table>
<thead>
<tr>
<th>Model</th>
<th>Characteristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Threads</td>
<td>Parallelism, blocking system calls</td>
</tr>
<tr>
<td>Single-threaded process</td>
<td>No parallelism, blocking system calls</td>
</tr>
<tr>
<td>Finite-state machine</td>
<td>Parallelism, nonblocking system calls, interrupts</td>
</tr>
</tbody>
</table>

Three ways to construct a server
Common thread programming models

- **Manager/worker**
  - Manager thread handles I/O and assigns work to worker threads
  - Worker threads may be created dynamically, or allocated from a thread-pool

- **Pipeline**
  - Each thread handles a different stage of an assembly line
  - Threads hand work off to each other in a producer-consumer relationship
What does a typical thread API look like?

- **POSIX standard threads (Pthreads)**
- First thread exists in main(), typically creates the others

- **pthread_create (thread, attr, start_routine, arg)**
  - Returns new thread ID in “thread”
  - Executes routine specified by “start_routine” with argument specified by “arg”
  - Exits on return from routine or when told explicitly
Thread API (continued)

- **pthread_exit (status)**
  - Terminates the thread and returns “status” to any joining thread

- **pthread_join (threadid, status)**
  - Blocks the calling thread until thread specified by “threadid” terminates
  - Return status from pthread_exit is passed in “status”
  - One way of synchronizing between threads

- **pthread_yield ()**
  - Thread gives up the CPU and enters the run queue
Using create, join and exit primitives
An example Pthreads program

```c
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5

void *PrintHello(void *threadid)
{
    printf("\n%d: Hello World!\n", threadid);
    pthread_exit(NULL);
}

int main (int argc, char *argv[])
{
    pthread_t threads[NUM_THREADS];
    int rc, t;
    for(t=0; t<NUM_THREADS; t++)
    {
        printf("Creating thread %d\n", t);
        rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
        if (rc){
            printf("ERROR; return code from pthread_create() is %d\n", rc);
            exit(-1);
        }
    }
    pthread_exit(NULL);
}
```

Program Output

Creating thread 0
Creating thread 1
0: Hello World!
1: Hello World!
Creating thread 2
Creating thread 3
2: Hello World!
3: Hello World!
Creating thread 4
4: Hello World!

For more examples see: http://www.llnl.gov/computing/tutorials/pthreads
Pros & cons of threads

- **Pros**
  - Overlap I/O with computation!
  - Cheaper context switches
  - Better mapping to shared memory multiprocessors

- **Cons**
  - Potential thread interactions
  - Complexity of debugging
  - Complexity of multi-threaded programming
  - Backwards compatibility with existing code
User-level threads

- Threads can be implemented in the OS or at user level
- **User level thread implementations**
  - thread scheduler runs as user code
  - manages thread contexts in user space
  - OS sees only a traditional process
Kernel-level threads

The thread-switching code is in the kernel
User-level threads package

The thread-switching code is in user space
User-level threads

- **Advantages**
  - cheap context switch costs among threads in the same process!
  - User-programmable scheduling policy

- **Disadvantages**
  - How to deal with blocking system calls!
  - How to overlap I/O and computation!
Concurrent programming

Assumptions:
- Two or more threads
- Each executes in (pseudo) parallel
- We can’t predict exact running speeds
- The threads can interact via access to a shared variable

Example:
- One thread writes a variable
- The other thread reads from the same variable
- Problem - non-determinism:
  - *The relative order of one thread’s reads and the other thread’s writes determines the result!*
Race conditions

- **What is a race condition?**
  - two or more threads have an inconsistent view of a shared memory region (i.e., a variable)

- **Why do race conditions occur?**
  - values of memory locations replicated in registers during execution
  - context switches at arbitrary times during execution
  - threads can see “stale” memory values in registers
Counter increment race condition

- Incrementing a counter (load, increment, store)
- Context switch can occur after load and before increment!
Race Conditions

- **Race condition:** whenever the output depends on the precise execution order of the processes!

- **What solutions can we apply?**
  - prevent context switches by preventing interrupts
  - make threads coordinate with each other to ensure **mutual exclusion** in accessing **critical sections** of code
Mutual exclusion conditions

- No two processes simultaneously in critical section
- No assumptions made about speeds or numbers of CPUs
- No process running outside its critical section may block another process
- No process must wait forever to enter its critical section
Critical sections with mutual exclusion
How can we enforce mutual exclusion?

- What about using *locks*?
- Locks solve the problem of exclusive access to shared data.
  - Acquiring a lock prevents concurrent access
  - Expresses intention to enter critical section

- **Assumption:**
  - Each shared data item has an associated lock
  - Every thread sets the right lock before accessing shared data!
  - Every thread releases the lock after it is done!
Acquiring and releasing locks

Thread A

Thread B

Thread C

Thread D

Free

Lock
Acquiring and releasing locks

Thread A

Thread B

Thread C

Thread D

Lock

Free

Lock
Acquiring and releasing locks

Thread A

Thread B

Thread C

Thread D

Set

Lock

Lock
Acquiring and releasing locks

Thread A

Lock

Set

Thread B

Thread C

Thread D
Acquiring and releasing locks

Thread A

Thread B

Thread C

Thread D

Set

Lock
Acquiring and releasing locks

Thread A

Thread B

Lock

Thread C

Thread D

Set

Lock
Acquiring and releasing locks

Thread A

Thread B

Lock

Thread C

Set

Lock

Thread D
Acquiring and releasing locks

Thread A

Thread B → Lock

Thread C → Lock

Thread D → Lock

Set Lock
Acquiring and releasing locks

Thread A

Thread B

Thread C

Thread D

Set

Lock

Lock

Lock

Lock
Acquiring and releasing locks

Thread A

Thread B

Thread C

Thread D

Set

Unlock

Lock

Lock

Lock

Lock
Acquiring and releasing locks

Thread A

Thread B

Unlock

Thread C

Lock

Thread D

Set

Unlock
Acquiring and releasing locks
Acquiring and releasing locks

Thread A

Thread B

Thread C

Thread D

Free

Lock

Lock

Lock

Lock
Acquiring and releasing locks

Thread A

Thread B

Thread C

Thread D

Set

Lock

Lock

Lock

Lock
Acquiring and releasing locks

Thread A

Thread B

Thread C

Thread D

Set

Lock

Lock

Lock

Lock
Acquiring and releasing locks

Thread A

Thread B

Thread C

Thread D

Lock

Set

Lock

Lock

Lock
Mutex locks

- An abstract data type
- Used for synchronization and mutual exclusion
- The mutex is either:
  - Locked ("the lock is held")
  - Unlocked ("the lock is free")
Mutex lock operations

- **Lock** (*mutex*)
  - Acquire the lock if it is free
  - Otherwise wait until it can be acquired

- **Unlock** (*mutex*)
  - Release the lock
  - If there are waiting threads wake up one of them
How to use a mutex?

*Shared data:*

```c
Mutex myLock;
```

1 repeat
2 `Lock(myLock);`
3 `critical section`
4 `Unlock(myLock);`
5 `remainder section`
6 until FALSE

1 repeat
2 `Lock(myLock);`
3 `critical section`
4 `Unlock(myLock);`
5 `remainder section`
6 until FALSE
How to implement a mutex?

- Both **Lock** and **Unlock** must be *atomic*!
  - Does a binary “lock” variable in memory work?

- Many computers have *some limited* hardware support for setting locks
  - Atomic Test and Set Lock instruction
  - Atomic compare and swap operation

- Can be used to implement mutex locks
Test-and-set-lock instruction (TSL, tset)

- A lock is a single word variable with two values
  - 0 = FALSE = not locked
  - 1 = TRUE = locked

- Test-and-set does the following *atomically*:
  - Get the (old) value
  - Set the lock to TRUE
  - Return the old value

If the returned value was FALSE...
Then you got the lock!!!

If the returned value was TRUE...
Then someone else has the lock
(so try again later)
Test and set lock

P1

FALSE

Lock
Test and set lock

FALSE = Lock Available!!

FALSE

P1

Lock
Test and set lock

P1

FALSE

TRUE

Lock
Test and set lock
Test and set lock
Test and set lock

P1

P2

TRUE

TRUE

P3

TRUE

TRUE

P4

TRUE

TRUE

TRUE

TRUE

Lock
Test and set lock

P1

P2

FALSE

Lock

P3

TRUE

P4

FALSE
Test and set lock
Test and set lock

P1

P2

P3

P4

TRUE

FALSE

TRUE

Lock
Test and set lock

P1

P2

P3

P4

TRUE

TRUE

TRUE

TRUE

TRUE

Lock
Critical section entry code with TSL

1 repeat I
2 while(TSL(lock))
3 no-op;
4 critical section
5 Lock = FALSE;
6 remainder section
7 until FALSE

1 repeat J
2 while(TSL(lock))
3 no-op;
4 critical section
5 Lock = FALSE;
6 remainder section
7 until FALSE

- Guarantees that only one thread at a time will enter its critical section
- Note that processes are **busy** while waiting
  - Spin locks
Busy waiting

- Also called polling or spinning
  - The thread consumes CPU cycles to evaluate when the lock becomes free!

- Shortcoming on a single CPU system...
  - A busy-waiting thread can prevent the lock holder from running & completing its critical section & releasing the lock!
  - Why not block instead of busy wait?
Quiz

- What is the difference between a program and a process?
- Is the Operating System a program?
- Is the Operating System a process?
  - Does it have a process control block?
  - How is its state managed when it is not running?
- What is the difference between processes and threads?
- What tasks are involved in switching the CPU from one process to another?
  - Why is it called a context switch?
- What tasks are involved in switching the CPU from one thread to another?
  - Why are threads “lightweight”?