Administrivia

• Project Discussion
  – Check out the example program that uses fork and pipes
  – Will give you some help with the bonus portion starting today

• Last Time:
  – Process Management/Inter Process Communication
  – Started Threads
Cooperating Processes

• Independent processes are not affected by the execution of other processes

• However, an application may be organized as a set of cooperating processes – why?
  – Information sharing (e.g., shared file)
  – Computation speedup – how?
  – Modularity
  – Convenience (e.g., the command line interpreter spawning multiple tasks)

• OS provides several facilities to support this cooperation

• We looked at a Producer consumer problem to illustrate solutions
OS IPC Facilities – Unix as a typical Example

• Signals (software interrupt): one process signals another
  – Signalled process executes handler for the signal
  – Signals can be ignored (signal mask)
  – Handlers can be changed

• Shared memory
  – shmget, shmat, shmdt
  – Have to manage shared variables carefully

• Message passing
  – One to one (e.g., pipes, message queues and sockets)
  – many to many (e.g., named pipes/FIFOs)
Why Threads?

- Decouple “thread of execution” from “resources” roles of a process

- A thread is a schedulable entity – why not just use processes?
  - More efficient:
    - Significantly lower overhead for creating threads, destroying them, and switching among them
  - More appropriate than processes to execute different activities operating on the same set of objects

- Is there an advantage to sharing the resources (memory, files, etc..)?

- Is there a disadvantage?
Threads – Nuts and Bolts

- A thread consists of:
  - Program counter
  - Register set
  - stack

- A thread shares with its peer threads
  - Code section
  - Data section
  - OS resources (files etc..)
Threads Implementation

- Threads can be
  - Kernel level (or lightweight processes)
    * Implemented by the OS; they are schedulable by the kernel
  - User level
    * Implemented by the user (or user-level library)
    * A thread scheduler runs within the user process and does the scheduling
  - What are the advantages/disadvantages?

- How are thread states different from process states?
Kernel Level vs. User Level Threads

- Kernel Level threads (one user thread to one kernel thread)
  - Threads are visible to the kernel and managed by it

- User Level threads (many user threads to one kernel thread)
  - Threads Managed in user space
- Many to Many?
Kernel Level Threads

Kernel Level Threads

User Level

Kernel Level

thread 1

thread 2

interrupt

ready

running

I/O or
event complete

scheduler dispatch

waiting

I/O or event wait

interrupt

ready

running

I/O or event complete

scheduler dispatch

waiting

I/O or event wait

Kernel Level Thread

kernel scheduler
User Level Threads

User Level Thread
- **interrupt**
- **I/O or event complete**
- **scheduler dispatch**
- **waiting**
- **running**

Kernel Level Thread
- **interrupt**
- **I/O or event complete**
- **scheduler dispatch**
- **waiting**
- **running**
Discussion

- What happens when a user level thread blocks on I/O
  - User level threads implementation
  - Kernel level threads implementation

- Is there any advantage to user level threads??
Relative Costs of Different Alternatives

- Operation Latencies in $\mu$-seconds

<table>
<thead>
<tr>
<th>Operation</th>
<th>User Level Threads</th>
<th>Kernel Level Threads</th>
<th>Process</th>
</tr>
</thead>
<tbody>
<tr>
<td>Null Fork</td>
<td>34</td>
<td>948</td>
<td>11,300</td>
</tr>
<tr>
<td>Signal Wait</td>
<td>37</td>
<td>441</td>
<td>1,840</td>
</tr>
</tbody>
</table>

- Null fork: time to create, schedule, execute and complete a process/thread that executes an empty procedure

- Signal wait: the time for a process/thread to signal a waiting process/thread and then wait on a condition

- For reference, procedure call cost 7, kernel trap 17 $\mu$-seconds
Many to Many (user threads to kernel threads)

- Any user level thread can be executed by any kernel level thread
Case Study: Solaris 2 Threads

- Solaris 2 is a version of UNIX with support for threads at the kernel and user levels, symmetric multiprocessing and real-time scheduling

- User level threads
  - Good: low-overhead for creating and switching
  - Bad: if one thread blocks, the whole “process” blocks – only one thread at the kernel level

- Kernel level threads
  - Good: if one thread blocks, the OS can schedule another
  - Bad: more expensive to create and switch (need a mode switch to kernel)

- Hybrid philosophy, combine the best of both
Threading Odds and Ends

- Cancelling threads: one thread eliminating another thread – why?
  - Immediate cancellation: kill it now
  - Deferred cancellation: tell it to kill itself, it will do it later when it checks
- Problem: signals are delivered to a process; which thread should get them?
- Thread Pools: have a pool of available threads, they are assigned to tasks as they come in (e.g., on a threaded web server)
  - More efficient than creating a thread from scratch
  - Limits the number of threads in the system (otherwise they can grow indefinitely)
POSIX Thread Model

```c
#include <pthread.h>
#include <errno.h>
```

- `sterror(errno)` returns a text description of `errno`

- Creating a thread:
  ```c
  int pthread_create(pthread_t * thread,
                    const pthread_attr_t * attr,
                    void *(*start)(void *),
                    void *arg);
  ```

- Thread descriptor is pointed to by `thread`

- Thread attributes specify: detach state, stack size/address, scheduling policy, scheduler parameters, etc... (use NULL to get default)

- Third parameter is a pointer to a procedure that the thread starts executing at – notice signature

- Fourth parameter is argument list to pass to the procedure
#include <pthread.h>
#include <stdio.h>

int num = 0;

void *add_one(int * thread_num) {
    num++;
    printf("thread %d  num = %d\n",  *thread_num, num);
}

void main() {
    pthread_t *thread;
    int my_id = 0;
    int your_id = 1;
    pthread_create(thread, NULL, add_one, &your_id);
    add_one(&my_id);
    // pthread_join(*thread, NULL);
    pthread_exit(NULL);
}

• compile: gcc mythread.cc -o mythread -lpthread

• What is the output of this program?
A Closer Look

sethi %hi(num),%o1 ; put address of num in reg. o1
ld [%o1+%lo(num)],%o2; read num into register o2
add %o2,1,%o1 ; o1 = o2 + 1
st %o1,[%o0+%lo(num)]; store o1 back to num

sethi %hi(num),%o2 ; put address of num in reg. o2
ld [%o1],%o1
ld [%o2+%lo(num)],%o2; read num into o2
call printf,0 ; call printf

- portion of the add_one assembly (obtained using gcc -S mythread.cc and looking at mythread.s)

- Timer interrupt can happen after any instruction (switching to another thread)

- What are the possible outputs?
Possible Outputs

- A possible output (thread 0 num = 1; thread 1 num = 2)

- Other orders can produce (1 1; 0 2), (0 2; 1 1), (1 2; 0 1)

- Are these possible? (0 1; 1 1), (1 1; 0 1), (0 2; 1 2), (1 2; 0 2)

- Nondeterministic results – Headache! Wrong results – even worse!

- We need to organize access to shared variables

- atomic operations – operations guaranteed to execute without interference

- Use synchronization primitives to build up atomic sequences of instructions
Concurrency

• As you should be painfully aware of by now, problems can arise when there is concurrency

• Concurrency – multiple activities share a common data area or set of resources

• Is this a problem in:
  – Single Programmed Systems?
  – Multiprogrammed systems (threads? processes?)
  – Multiprocessor systems?

• Uncontrolled concurrency can give correct results, wrong results, or disastrous results

• Other problems: deadlock, livelock, and starvation

• Concurrency control crosses between the domains of OS and programming languages
So, what exactly is the problem?

- Problem is related to multiple threads accessing a shared variable.
  - In the `add_one` example, the variable `num` is the problem.

- We would like to control access to the shared variables.
  - Make sure that no other thread can access the variable in an inconsistent way.
  - For the `add_one` program, make sure the `num++` executes in one shot.
Example

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

int num = 0;
pthread_mutex_t num_mutex = PTHREAD_MUTEX_INITIALIZER;

void *add_one(int * thread_num) {
    pthread_mutex_lock(&num_mutex);
    num++;
    printf("thread %d num = %d\n", *thread_num, num);
    pthread_mutex_unlock(&num_mutex);
}

void main() {
    pthread_t *thread;
    int my_id = 0;
    int your_id = 1;
    pthread_create(thread, NULL, add_one, &your_id);
    add_one(&my_id);
    pthread_exit(NULL);
}
```

- What are the possible outcomes now?
while(1) {
    ...
    entry section //getting the lock
    critical section
    exit section // releasing the lock
    ...
}

• Problem Description:
  – $n$ processes competing to use shared data
  – Portions of the code that use the shared data are called critical sections
  – Problem: ensure only one process in the critical section

• An acceptable solution should:
  1. Ensure Mutual Exclusion (at most one process in the critical region)
  2. Ensure Progress is made (if region is empty, and there are processes that need it, they should be able to enter)
  3. Ensure no Starvation (after a process arrives, there is a bound on the number of processes that go in before it)
How to Implement Locks – Software Approaches

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

Process 0, 1
.
.
while(!pthread_trylock(mutex));
<critical region>
pthread_unlock(mutex);

- Fictious implementation of trylock – does it work?
- What is the fundamental problem?
First Attempt: Better Solution

```c
bool turn;

Process 0               Process 1

.                      .
.                      .
while (turn != 0);     while (turn != 1);
[Critical Section]     [Critical Section]
turn = 1;              turn = 0;
```

- Does this work?
- Which of the requirements are not satisfied?
- Drawbacks?
  - Strictly alternating order; may not map well to application needs
  - What if there is more than two?
  - What if a process fails?
bool flag[2];

Process 0
.
.
while (flag[1] != 0);
flag[0] = 1;
[Critical Section]
flag[0] = 0;

Process 1
.
.
while (flag[0] != 0);
flag[1] = 1;
[Critical Section]
flag[1] = 0;

• Problem Solved?
  – Strict turns do not have to be followed
  – Process failure still a problem?

• Is starvation a problem?

• Wrong Solution – why?