Midterm Preliminaries

- Lectures 1–9 (excluding scheduling)
- How to prepare for the midterm?
  - Previous years’ midterm on class webpage – try to do some of these
  - Do problems in book
  - As you read, do so critically (ask yourself why, how, can it be done better?)

Midterm Preliminaries (cont’d)

- Different levels of questions
- Some questions will be a little difficult
  - Will ask you to apply principles in a new situation and have an understanding of the basic tradeoffs and alternatives
    - Prepare for this
    - Some questions may be open: more than one answer is possible
- Midterm will probably be a little long
  - Work quickly; try to avoid wasting a lot of time on one problem
  - Short explanations with the main points
  - Try to attempt everything (can’t give partial credit for no answer)
A tour of Computer Organization

- A Computer System consists of:
  - Processor: the workhorse
    - Arithmetic Logic Unit (the number cruncher)
    - General Purpose Registers (data/address)
    - Control Registers (PC, IR, etc.)
    - Control unit
  - Memory: holds instructions and data
  - I/O Devices: interface with the world

- How a program runs:
  - Your program is compiled into a machine language program
  - The processor executes a sequence of machine instructions
  - Instructions are encoded in the machine specific instruction format

Key Ideas for OS

1. The CPU must also be able to handle interrupts
   - After the fetch, decode, execute, check if there is an interrupt
     - If there is, branch into the interrupt service routine

2. The CPU must provide a protected execution mode
   - Why? Cannot have users directly access resources or other processes memory locations
   - OS only runs in the priviledged mode

I/O Devices

- An I/O device usually has a controller – a small processor with some buffering space
- Actual I/O happens between device and the controller buffering space, it is then copied to memory
- Programmed I/O (read):
  - Processor issues I/O command to device
  - Keeps checking status until device is done (or failed), then copies data to local memory
- Interrupt Driven I/O
  - Rely on the I/O device to signal an interrupt when it is done
  - The processor still carries out the memory to memory transfer
- Direct Memory Access
  - A separate controller carries out the I/O operation for the CPU
  - What if the processor needs the memory while the DMA is underway?
System calls

- Provides the interface between a running program and the OS
  - how does it compare to a command-line interpreter?
  - Generally assisted by an assembly language instruction (trap, or syscall)
  - How are they accessed from programming languages?

- Implementation wise, a system call is like a procedure call to an OS procedure
  - Passing the parameters?
    * In registers
    * In a table in memory; pass a pointer as the only parameter
    * Use a stack
  - Need a **mode switch** to the privileged mode
  - Can you give specific examples of system calls?

Process Management

- Process is
  - A program during execution
  - Unit of resource management
- Key concept: processes are completely separate — a process cannot directly affect the state of another
- Stages in the Life of a Process
  - Birth: Process Creation – How is a process created?
    * Created by the OS to provide a service
    * Spawned (forked) by an existing process
  - Death: Process Termination – How/Why does a process die?
    * Normal Termination (its task is done)
    * Errors, abuse of privileges, parent request/parent death, etc..
  - In between the process lives in one of the following stages: (1) Running; (2) Waiting; (3) Ready

Process Control Block (PCB)

- OS maintains information about each process in a data structure called a Process Control Block (PCB)
- Process image: program, data, stack and PCB
- PCB includes
  - Process Identification information: process id, etc..
  - Process state
    * Program counter value
    * Registers
  - Process Control Information
    * CPU scheduling information
    * Memory management information
    * Accounting information
    * I/O status information
    * File information

Process Scheduling

- A “dispatcher” (also called “scheduler”) runs after each process, and decides which process gets to run
Scheduling Queues

- Job queue – set of all processes in the system
- Ready queue
- Device queues
- How does a process move among queues?

Process Scheduling

- Short-Term Scheduler decides which process to run next
- Long-Term Scheduler decides which new process can be brought into the ready queue
  - Controls the degree of multiprogramming of the system
- Medium Term Scheduler – Gives us a chance to reverse decisions made by Long term scheduler

Medium Term Scheduler

- More often than long-term, but less than short-term
- A process that is **swapped out** is removed from memory and is not eligible to run
- A process that is **swapped in** is brought back into memory
- Why/How is this useful?
- How does the process state diagram look now?
Threads vs. Processes

- Two aspects to the process definition:
  - A stream of execution (in a thread state) - allowing multiple threads to reside in the same process
  - A thread or a light-weight process is a stream of execution (in a thread state) - allowing multiple threads to reside in the same process

- The two aspects are independent - allowing multiple processes to execute in own process state (resource management unit) or no

- Why is decoupling the two concepts useful?

- A thread or a light-weight process is a stream of execution (in a thread state) - allowing multiple threads to reside in the same process

Kernel Level Threads vs. User Level Threads

- Many to Many?
  - User Level threads (many user threads to one kernel thread)
  - Kernel Level threads (one user thread to one kernel thread)

- Threads are visible to the kernel and managed by:
  - User Level threads
  - Kernel Level threads

- Threads Managed in user space

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

- A thread consists of:
  - Program counter
  - Register set
  - Stack
  - Code section
  - Data section
  - OS resources (files etc.)

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

Threads - Nuts and Bolts
With Concurrency come problems

```c
#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

```assembly
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
```

```assembly
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!
- Problem is related to multiple threads accessing a shared variable
- atomic operations – operations guaranteed to execute without interference
- Use synchronization primitives to build up atomic sequences of instructions
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);
}
```

The Critical Section Problem

```c
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 that at most one process is 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 at the region, there is a bound on the number of processes that will be allowed to go in before it)

How to Implement Locks – Software Approaches

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

- Fictitious implementation of trylock – does it work?
- What is the fundamental problem?

Correct Implementation: Dekker’s Algorithm

```c
bool flag[2];
int turn = 0;

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

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

[Critical Section]
[Critical Section]
flag[0] = 0; flag[1] = 0;
turn = 1; turn = 0;
```

- The two flags solve the mutual exclusion problem; use the turn (as per the first implementation) to solve simultaneous interest problem
- Do we have the alternating execution problem?
More Elegant Solution: Peterson’s Algorithm

```c
bool flag[2];
int turn = 0;

Process 0                Process 1
.                      .
flag[0] = 1;            flag[1] = 1;
turn = 1;               turn = 0;
while (flag[1] == 1 && turn == 1);    while (flag[0] == 1 && turn == 0);
[Critical Section]      [Critical Section]
flag[0] = 0;            flag[1] = 0;
```

- Does this work? How?
- Is it fair (starvation/alternating execution?)
- How can we prove its correctness?

Bakery Algorithm

```c
//choosing, ticket are shared
...
choosing[i] = TRUE;
ticket[i] = max (ticket[0], ticket [1] ...
ticket [n]) + 1;
choosing[i] = FALSE;
for(j = 0; j < n; j++) {
    while (choosing[j] == TRUE);
    while (ticket[j] != 0 && (ticket[j],j) < (ticket [i],i));
}    
[Critical Section]
ticket[i] = 0;
...

• (ticket[j],j) < (ticket[i],i) refers to the comparison including using the process number as tie-breaker if tickets equal
• Take your time, think about it
• Does it satisfy the three requirements?
```

Hardware Assist: Test-and-Set

```c
bool lock = 0;

Process 0                Process 1
.                      .
while (testAndSet(lock)); while (testAndSet(lock));
[Critical Section]      [Critical Section]
lock = 0;               lock = 0;
```

- Simpler
- Still busy waits
- Generalizes to any number of processes/locks
- What are the implications if used on a Shared Memory Multiprocessor?
- Is waiting bounded?
- Example of test-and-op class of primitives

Test and Set for $n$ Processes with Bounded Wait

```c
waiting[i] = 1;
key = 1;
while(waiting[i] && key)
    key = testAndSet(lock);
waiting[i] = 0;

[Critical Section]

j = i+1 % n
while (((j != i) && !waiting[j])
    j = j + 1 % n;
if (j == i)
    lock = 0;
else
    waiting[j] = 0;
```

- Easier than bakery algorithm?
Semaphores

- Informally, multi-valued, queued, blocking lock
- Semaphore consists of a value and a queue
- Acquiring a Semaphore (aka, wait, down, or P):
  - If value is positive, decrement it and let the process in
  - If value is zero, put the process at the end of the queue
- A process releases a semaphore (aka, signal, up, or V)
  - If queue is not empty, head of the queue is allowed to acquire S
  - If the queue is empty, increment S’s value
- Access to the semaphore information should be atomic (how?/why?)
- Can be used to enforce mutual exclusion as well as synchronization

Implementation

```c
void wait(Semaphore S) {
    Lock(S.Lock); // added Lock variable to Semaphore
    if (S.value == 0)
        Add to queue and block; // proceed when unblocked
    else
        S.value--;
    Unlock(S.Lock);
}

void signal (Semaphore S) {
    Lock(S.Lock);
    if (!empty(S.q))
        Unblock Top of queue;
    else
        S.value++;
    Unlock(S.Lock);
}
```

- Lock and Unlock can be any of our software locks, or even disabling interrupts
- Is there a problem in this implementation? think about what happens when a process blocks

Correct Implementation Using Binary Semaphores

```c
// Binary semaphore Mutex, initialized to 1
// empty, initialized to 0
int n = 0, m;

// Producer
while (1) {
    produce();
    wait(mutex);
    buffer.append(nextp);
    signal(mutex);
    signal(full);
    if(n == 1)
        m = n;
    signal(empty);
}

// Consumer
while(1) {
    wait(full);
    wait(mutex);
    nextc = buffer.nextItem();
    signal(mutex);
    signal(empty);
}
```

- Why does this additional assignment solve anything?
- A little clumsy

Bounded Buffer Producer Consumer

```c
// Initial values: mutex = 1, full = 0, empty = n
Producer:
while (1) {
    produce an item in nextp;
    wait(empty);
    wait(mutex);
    buffer.append(nextp);
    signal(mutex);
    signal(full);
}

Consumer:
while(1) {
    wait(full);
    wait(mutex);
    nextc = buffer.nextItem();
    signal(mutex);
    signal(empty);
}
```
Solution to Readers Writers Problem

//Semaphore mutex initialized to 1
//Semaphore write initialized to 1
readers
writers

wait(mutex);
readers++;
if(readers == 1)
WRITE;
wait(write);
signal(mutex);

READ;
wait(mutex);
readers--;
if(readers == 0)
signal(write);
signal(mutex);

• Problems?

Producer Consumer – Condition Locks

Condition C;
Lock condLock;
------------Producer-------------
while (1) {
produce an item in nextp;
Lock(condLock);
if(count == MAX)
C.wait(condLock);
count++;
buffer.append(nextp);
if(count == 1)
C.signal(condLock);
Unlock(condLock); }
------------Consumer-------------
while(1) {
Lock(condLock);
if(count == 0)
C.wait(condLock);
nextc = buffer.nextItem();
if(count==MAX)
C.signal(condLock);
count--;
Unlock(condLock);
Consume nextc;
}

Bounded Buffer

• Definition:

var buffer: shared record
pool array [0...n-1] of item
  count, in, out: integer;
end;

• Producer:

produce nextp;
region buffer when count < n
  pool[in] = nextp;
in = in + 1 mod n;
count = count + 1;
end;

High Level Constructs for Mutual Exclusion

• General Idea: give help to the programmer by making mutual exclusion automatic on shared data

• Conditional Critical Regions

• Monitors
Monitor prod_cons
Condition empty; full;
int buffer[SIZE], count, in, out;
void append(int item) {
    if(count == SIZE)
        empty.wait();
    count++;
    buffer[in] = item;
in = (in + 1) % SIZE;
    if(count == 1)
        full.signal();
}
int get() {
    int nextc; //local variable
    if(count == 0)
        full.wait();
    nextc = buffer[out];
    out = (out + 1) % size;
    if(count == SIZE)
        empty.signal();
    count--;
    return nextc;
}
begin
    in = out = count = 0;
end;

//Producer
while(1) {
    nextp = produce();
    prod_cons.append(nextp);
}

//Consumer
while(1) {
    nextc = prod_cons.take();
    consume(nextc);
}

Problem 2 – Spring 00 Midterm
(a) (9 points) An operating system is replete with queues. Name three
different queues that a process may belong to; under what situation/state
would you expect the process to be there? (Name 3 more for a 3 point
bonus)
(b) (11 points) Consider the following program fragments being executed
by two threads; what is the range of final values that x can have
Thread 1
for(i=0; i < 10; i++)
    x = x + 2;

Thread 2
for(j=0; j < 10; j++)
x = x - 1;

Problem 3: Spring 00 Midterm
Briefly explain any 4 of the following potentially wrong statements. Be
careful with this problem.
1. Busy waiting is more expensive than blocking
2. It is beneficial to have "user-level" processes in the same way we have
   "user-level" threads
3. Mesa and Hoare semantics in monitors make no difference to the
   programmer

Solution to Readers Writers Problem
//Semaphore mutex initialized to 1
//Semaphore write initialized to 1
readers
writers
wait(mutex);
readers++;
if(readers == 1) WRITE;
wait(write);
signal(mutex);
signal(mutex);
READ;
wait(mutex);
readers--;
signal(mutex);

• Problems?
**Problem 4: Final 00**

Consider the case of passengers standing in the check-in line at the airport. Every passenger stands in line until they are at the top of the line. There are 5 airline employees. Each one of them takes care of a passenger, then calls for the next one when they are free.

(a) (8 points) Write the pseudo-code simulating the passengers and the airline employees using semaphores and/or condition locks. Be careful of the case when multiple employees become free at the same time.

(b) (7 points) First class passengers get preferential treatment and do not stand in line. Update your implementation to account for first class passengers.

**Problem 6: Spring 03**

Consider the following versions of the producer consumer problem. They differ in the mechanism used for implementing the critical region as well as what is included in the critical region: (1) A version that uses Peterson’s algorithm to restrict access around the shared buffer; (2) A version that disables interrupts around the access to the shared buffer; (3) A version that uses a binary semaphore around the access of the shared buffer; and (4) a version that uses semaphores around the full consumer and producer code.

(a) (8 points) Show an implementation for version (3) with an unbounded buffer.

(b) (9 points) Explain which of these versions will be most efficient.