Overview

- Shared memory machines
- Programming strategies for shared memory machines
- Allocating shared data for IPC
- Processes and threads
- MT-safety issues
- Coordinated access to shared data
  - Locks
  - Semaphores
  - Condition variables
  - Barriers
- Further reading
Shared Memory Machines

- Single address space
- Shared memory
  - Single bus (UMA)
  - Interconnect with memory banks (NUMA)
  - Cross-bar switch
- Distributed shared memory (DSM)
  - Logically shared, physically distributed

Shared memory UMA machine with a single bus

Shared memory NUMA machine with memory banks

DSM

Shared memory multiprocessor with cross-bar switch
Programming Strategies for Shared Memory Machines

- Use a completely new programming language for parallel computing
  - For example: HPF, UPC
- Use compiler directives to supplement a sequential program with parallel directives
  - For example: OpenMP
- Use library routines with a sequential program
  - For example: ScaLapack (though ScaLapack is primarily designed for distributed memory)
- Use heavyweight processes and a shared memory API
- Use threads
- Use a parallelizing compiler to transform (part of) a sequential program into a parallel program
Heavyweight Processes

- The UNIX system call `fork()` creates a new process
  - `fork()` returns 0 to the child process
  - `fork()` returns process ID (pid) of child to parent
- System call `exit(n)` joins child process with parent and passes exit value `n` to it
- Parent executes `wait(&n)` to wait until one of its children joins, where `n` is set to the exit value
- System and user processes form a tree
Fork-Join

Process 1

```c
...  
...  
pid = fork();  
if (pid == 0)  
{ ... // code for child  
  exit(0);  
} else  
{ ... // parent code continues  
  wait(&n); // join  
}  
... // parent code continues  
...  
```

SPMD program
Fork-Join

Process 1

```c
... 
... pid = fork();
if (pid == 0)
{  ... // code for child
    exit(0);
} else
{  ... // parent code continues
    wait(&n); // join
}
... // parent code continues
...```

Process 2

```c
... 
... pid = fork();
if (pid == 0)
{  ... // code for child
    exit(0);
} else
{  ... // parent code continues
    wait(&n); // join
}
... // parent code continues
...```

SPMD program

Copy of program, data, and file descriptors (operations by the processes on open files will be independent)
Fork-Join

Process 1

... 
... 
pid = fork();  
if (pid == 0) 
{ ... // code for child 
  exit(0); 
} else 
{ ... // parent code continues 
  wait(&n); // join 
}
... // parent code continues 
...

Process 2

... 
... 
pid = fork();  
if (pid == 0) 
{ ... // code for child 
  exit(0); 
} else 
{ ... // parent code continues 
  wait(&n); // join 
}
... // parent code continues 
...

SPMD program

Copy of program and data
Fork-Join

Process 1

```c
...  
...  
pid = fork();  
if (pid == 0)  
{ ... // code for child  
  exit(0);  
} else  
{ ... // parent code continues  
  wait(&n); // join  
}  
... // parent code continues  
...  
```

SPMD program

Process 2

```c
...  
...  
pid = fork();  
if (pid == 0)  
{ ... // code for child  
  exit(0);  
} else  
{ ... // parent code continues  
  wait(&n); // join  
}  
... // parent code continues  
...  
```

Copy of program and data
Fork-Join

Process 1

```c
... 
... 
pid = fork();
if (pid == 0)
{ ... // code for child
  exit(0);
}
else
{ ... // parent code continues
  wait(&n); // join
}
... // parent code continues
... 
```

Process 2

```c
... 
... 
pid = fork();
if (pid == 0)
{ ... // code for child
  exit(0);
}
else
{ ... // parent code continues
  wait(&n); // join
}
... // parent code continues
... 
```

SPMD program

Terminated
Creating Shared Data for IPC

- **Interprocess communication (IPC)** via shared data
- Processes do not automatically share data
- Use files to share data
  - Slow, but portable
- Unix system V **shmget()**
  - Allocates shared pages between two or more processes
- BSD Unix **mmap()**
  - Uses file-memory mapping to create shared data in memory
  - Based on the principle that files are shared between processes

**shmget()**
returns the shared memory identifier for a given key (key is for naming and locking)

**shmat()**
attaches the segment identified by a shared memory identifier and returns the address of the memory segment

**shmctl()**
deletes the segment with `IPC_RMID` argument

**mmap()**
returns the address of a mapped object described by the file id returned by `open()`

**munmap()**
deletes the mapping for a given address
shmget vs mmap

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

size_t len;  // size of data we want
void *buf;  // to point to shared data
int shmid;
key_t key = 9876;  // or IPC_PRIVATE
shmid = shmget(key,
    len,
    IPC_CREAT|0666);
if (shmid == -1)  // error
    buf = shmat(shmid, NULL, 0);
if (buf == (void*)-1)  // error
    ...
    fork();  // parent and child use buf
    ...
    wait(&n);
    shmctl(shmid, IPC_RMID, NULL);

#include <sys/types.h>
#include <sys/mman.h>

size_t len;  // size of data we want
void *buf;  // to point to shared data
int shmid;
key_t key = 9876;  // or IPC_PRIVATE
shmid = shmget(key,
    len,
    PROT_READ|PROT_WRITE,
    MAP_SHARED|MAP_ANON,
    -1,  // fd=-1 is unnamed
    0);
if (buf == MAP_FAILED)  // error
    ...
    fork();  // parent and child use buf
    ...
    wait(&n);
    munmap(buf, len);
    ...

Tip: use **ipcs** command to display
IPC shared memory status of a system
Threads

- *Threads of control* operate in the same memory space, sharing code and data
  - Data is implicitly shared
  - Consider data on a thread’s stack private
- Many OS-specific thread APIs
  - Windows threads, Linux threads, Java threads, …
- POSIX-compliant Pthreads:
  - `pthread_create()`
    start a new thread
  - `pthread_join()`
    wait for child thread to join
  - `pthread_exit()`
    stop thread
Detached Threads

- Detached threads do not join
- Use \texttt{pthread_detach(thread_id)}
- Detached threads are more efficient
- Make sure that all detached threads terminate before program terminates
Process vs Threads

What happens when we fork a process that executes multiple threads?
Does fork duplicate only the calling thread or all threads?
Thread Pools

- *Thread pooling* (or *process pooling*) is an efficient mechanism
- One *master thread* dispatches jobs to worker threads
- *Worker threads* in pool never terminate and keep accepting new jobs when old job done
- Jobs are communicated to workers via shared data and/or *signals*
MT-Safety

Routines must be multithreaded-safe (MT-safe) when invoked by more than one thread

Non-MT-safe routines must be placed in a critical section, e.g. using a mutex lock (see later)

Many C libraries are not MT-safe
- Use libroutine_r() versions that are “reentrant”
- When building your own MT-safe library, use #define _REENTRANT

Always make your routines MT-safe for reuse in a threaded application

Use locks when necessary (see next slides)

Use of a not-MT-safe routine

```c
time_t clk = clock();
char *txt = ctime(&clk);
printf(“Current time: %s
”, txt);
```

Use of the reentrant version of ctime

```c
static int counter = 0;
int count_events()
{ return counter++; }
```

Is this routine MT-safe? What can go wrong?
Coordinated Access to Shared Data

- Reading and writing shared data by more than one thread or process requires coordination with *locking*.
- Cannot update shared variables simultaneously by more than one thread.

```c
static int counter = 0;
int count_events()
{
    return counter++;
}
```

```c
static int counter = 0;
int count_events()
{
    pthread_mutex_lock(&lock);
    counter++;
    pthread_mutex_unlock(&lock);
    return counter-1;
}
```

Thread 1

reg1 = M[counter] = 3
reg2 = reg1 + 1 = 4
M[counter] = reg2 = 4
return reg1 = 3

Thread 2

reg1 = M[counter] = 3
reg2 = reg1 + 1 = 4
M[counter] = reg2 = 4
return reg1 = 3

Thread 1

acquire lock
reg1 = M[counter] = 3
reg2 = reg1 + 1 = 4
M[counter] = reg2 = 4
release lock

Thread 2

acquire lock
…
… wait
…
…
reg1 = M[counter] = 4
reg2 = reg1 + 1 = 5
M[counter] = reg2 = 5
release lock
Spinlocks

- Spin locks use busy waiting until a condition is met
- Naïve implementations are almost always incorrect

// initially lock = 0

```c
while (lock == 1) {
    // do nothing
    lock = 1;
    ... critical section ...
    lock = 0;
}
```

Acquire lock

Release lock

Two or more threads want to enter the critical section, what can go wrong?
Spinlocks

- Spin locks use busy waiting until a condition is met
- Naïve implementations are almost always incorrect

Thread 1

```c
while (lock == 1) {
  ; // do nothing
}
lock = 1;
...
```

Thread 2

```c
while (lock == 1) {
  ...
}
lock = 1;
...
```

This ordering works
Spinlocks

- Spin locks use busy waiting until a condition is met
- Naïve implementations are almost always incorrect

Thread 1

```c
while (lock == 1) 
  ; // do nothing
lock = 1;
  ... critical section ...
lock = 0;
```

Thread 2

```c
while (lock == 1) 
  ...
lock = 1;
  ... critical section ...
lock = 0;
```

This statement interleaving leads to failure

Both threads end up executing the critical section!
Spinlocks

- *Spin locks* use *busy waiting* until a condition is met
- Naïve implementations are almost always incorrect

Thread 1

```
while (lock == 1)
    ; // do nothing
lock = 1;
    ... critical section ...
lock = 0;
```

Thread 2

```
    while (lock == 1)
    ... 
lock = 1;
    lock = 0;
    ... critical section ...
```

Compiler optimizes the code!

Useless assignment removed
Assignment can be moved by compiler

Atomic operations such as atomic “test-and-set” instructions must be used, and these instructions should not be allowed to be reordered or removed
Spinlocks

- Advantage of spinlocks is that the kernel is not involved
- Better performance when acquisition waiting time is short
- Dangerous to use in a uniprocessor system, because of priority inversion
- No guarantee of fairness and a thread may wait indefinitely in the worst case, leading to starvation

```c
void spinlock_lock(spinlock *s)
{ while (TestAndSet(s))
    while (*s == 1)
    ;
}

void spinlock_unlock(spinlock *s)
{ *s = 0;
}
```

Correct and efficient spinlock operations using atomic TestAndSet assuming hardware supports cache coherence protocol

Note: TestAndSet(int *n) sets n to 1 and returns old value of n
Semaphores

- A semaphore is an integer-valued counter
- The counter is incremented and decremented by two operations signal (or post) and wait, respectively
  - Traditionally called V and P (Dutch “verhogen” and “probeer te verlagen”)
- When the counter < 0 the wait operation blocks and waits until the counter > 0

```c
sem_post(sem_t *s)
{ (*s)++; }
```

```c
sem_wait(sem_t *s)
{ while (*s <= 0) ; // do nothing (*s)--;
 }
```

Note: actual implementations of POSIX semaphores use atomic operations and a queue of waiting processes to ensure fairness
Semaphores

- A two-valued (= binary) semaphore provides a mechanism to implement mutual exclusion (mutex)
- POSIX semaphores are named and have permissions, allowing use across a set processes

```
#include "semaphore.h"
sem_t *mutex = sem_open("lock371", O_CREAT, 0600, 1);
...
sem_wait(mutex); // sem_trywait() to poll state
...
... critical section ...
...
sem_post(mutex);
...
sem_close(mutex);
```

Tip: use `ipcs` command to display IPC semaphore status of a system
Pthread Mutex Locks

- POSIX mutex locks for thread synchronization
  - Threads share user space, processes do not

- Pthreads is available for Unix/Linux and Windows ports

```c
pthread_mutex_t mylock;

pthread_mutex_init(&mylock, NULL);
...
pthread_mutex_lock(&mylock);
... critical section ...
... pthread_mutex_unlock(&mylock);
...
pthread_mutex_destroy(&mylock);
```

- `pthread_mutex_init()` initialize lock
- `pthread_mutex_lock()` lock
- `pthread_mutex_unlock()` unlock
- `pthread_mutex_trylock()` check if lock can be acquired
Using Mutex Locks

- Locks are used to synchronize shared data access from any part of a program, not just the same routine executed by multiple threads.
- Multiple locks should be used, each for a set of shared data items that is disjoint from another set of shared data items (no single lock for everything).

```c
pthread_mutex_lock(&array_A_lck);
... A[i] = A[i] + 1 ... 
pthread_mutex_unlock(&array_A_lck);

pthread_mutex_lock(&queue_lck);
... add element to shared queue ... 
pthread_mutex_unlock(&queue_lck);

pthread_mutex_lock(&array_A_lck);
pthread_mutex_unlock(&array_A_lck);

pthread_mutex_lock(&queue_lck);
... remove element from shared queue ... 
pthread_mutex_unlock(&queue_lck);
```

Lock operations on array A

Lock operations on a queue

What if threads may or may not update some of the same elements of an array, should we use a lock for every array element?
Condition Variables

- **Condition variables** are associated with mutex locks
- Provide signal and wait operations *within* critical sections

`Process 1`

```plaintext
lock(mutex)
if (cannot continue)
  wait(mutex, event)
... 
unlock(mutex)
```

`Process 2`

```plaintext
lock(mutex)
... 
signal(mutex, event)
... 
unlock(mutex)
```

*Can’t use semaphore wait and signal here: what can go wrong?*
Condition Variables

**signal** releases one waiting thread (if any)

Process 1

```
lock(mutex)
if (cannot continue)
   wait(mutex, event)
...
unlock(mutex)
```

Process 2

```
lock(mutex)
...
signal(mutex, event)
...
unlock(mutex)
```

**wait** blocks until a signal is received

*When blocked, it releases the mutex lock, and reacquires the lock when wait is over*
Producer-Consumer Example

- Producer adds items to a shared container, when not full
- Consumer picks an item from a shared container, when not empty

A consumer

while (true)
{
    lock(mutex)
    if (container is empty)
        wait(mutex, notempty)
    get item from container
    signal(mutex, notfull)
    unlock(mutex)
}

A producer

while (true)
{
    lock(mutex)
    if (container is full)
        wait(mutex, notfull)
    add item to container
    signal(mutex, notempty)
    unlock(mutex)
}

Condition variables associated with mutex
Semaphores versus Condition Variables

- **Semaphores:**
  - Semaphores must have matching signal-wait pairs, that is, the semaphore counter must stay balanced
  - One too many waits: one waiting thread is indefinitely blocked
  - One too many signals: two threads may enter critical section that is guarded by semaphore locks

- **Condition variables:**
  - A signal can be executed at any time
  - When there is no wait, signal does nothing
  - If there are multiple threads waiting, signal will release one

- **Both provide:**
  - Fairness: waiting threads will be released with equal probability
  - Absence of starvation: no thread will wait indefinitely
Pthreads Condition Variables

- Pthreads supports condition variables
- A condition variable is always used in combination with a lock, based on the principle of "monitors"

**Declarations**

```
pthread_mutex_t mutex;
pthread_cond_t notempty, notfull;
```

**Initialization**

```
 pthread_mutex_init(&mutex, NULL);
pthread_cond_init(&notempty, NULL);
pthread_cond_init(&notfull, NULL);
```

**A consumer**

```
while (1)
{
    pthread_mutex_lock(&mutex);
    if (container is empty)
        pthread_cond_wait(&mutex, &notempty);
    get item from container
    pthread_cond_signal(&mutex, &notfull);
pthread_mutex_unlock(&mutex);
}
```

**A producer**

```
while (1)
{
    pthread_mutex_lock(&mutex);
    if (container is full)
        pthread_cond_wait(&mutex, &notfull);
    add item to container
    pthread_cond_signal(&mutex, &notempty);
pthread_mutex_unlock(&mutex);
}
```
Monitor with Condition Variables

- A monitor is a concept
- A monitor combines a set of shared variables and a set of routines that operate on the variables
- Only one process may be active in a monitor at a time
  - All routines are synchronized by implicit locks (like an entry queue)
  - Shared variables are safely modified under mutex
- Condition variables are used for signal and wait within the monitor routines
  - Like a wait queue

Only P1 executes a routine, P0 waits on a signal, and P2, P3 are in the entry queue to execute next when P1 is done (or moved to the wait queue)
Barriers

- A barrier synchronization statement in a program blocks processes until all processes have arrived at the barrier.
- Frequently used in data parallel programming (implicit or explicit) and an essential part of BSP.

*Each process produces part of shared data X*
*barrier*
*Processes use shared data X*
Two-Phase Barrier with Semaphores for $P$ Processes

\begin{verbatim}
sem_t *mutex = sem_open("mutex-492", O_CREAT, 0600, 1);
sem_t *turnstile1 = sem_open("ts1-492", O_CREAT, 0600, 0);
sem_t *turnstile2 = sem_open("ts2-492", O_CREAT, 0600, 1);
int count = 0;
...
sem_wait(mutex);
    if (++count == P)
    {
        sem_wait(turnstile2);
        sem_signal(turnstile1);
    }
sem_signal(mutex);
sem_wait(turnstile1);
sem_signal(turnstile1);
sem_wait(mutex);
    if (--count == 0)
    {
        sem_wait(turnstile1);
        sem_signal(turnstile2);
    }
sem_signal(mutex);
sem_wait(turnstile2);
sem_signal(turnstile2);
\end{verbatim}
Pthread Barriers

- Barrier using POSIX pthreads (advanced realtime threads)
- Specify number of threads involved in barrier syncs in initialization

```c
pthread_barrier_t barrier;

pthread_barrier_init(
    barrier,
    NULL,   // attributes
    count); // number of threads

...  

pthread_barrier_wait(barrier);

...  
```

```c
pthread_barrier_init()  
initialize barrier with thread count  

pthread_barrier_wait()  
barrier synchronization  
```
Further Reading

- [PP2] pages 230-247
- Optional:
  - [HPC] pages 191-218
  - “The Little Book of Semaphores” by Allen Downey
    http://www.greenteapress.com/semaphores/downey05semaphores.pdf