SE 292: High Performance Computing [3:0][Aug:2014]

Concurrent Programming

Yogesh Simmhan

Adapted from "Intro to Concurrent Programming & Parallel Arch.", Sathish Vadhiyar, SE292 (Aug:2013), "Operating Systems Concepts", Silberschatz, Galvin & Gagne, 2005 & Computer Systems: A Programmer's Perspective", by R.E. Bryant and D. O'Hallaron, 2003

Concurrent Programming

- Until now: execution involved one flow of control through program
- Concurrent programming is about programs with multiple flows of control
- For example: a program that runs as multiple processes cooperating to achieve a common goal
- To cooperate, processes must somehow communicate

Inter Process Communication (IPC)

1. Using files
   - Parent process creates 2 files before forking child process
   - Child inherits file descriptors from parent, and they share the file pointers
   - Can use one for parent to write and child to read, other for child to write and parent to read
2. OS supports something called a pipe
   - Producer writes at one end (write-end) and consumer reads from the other end (read-end)
   - corresponds to 2 file descriptors (int fd[2])
   - Read from fd[0] accesses data written to fd[1] in FIFO order and vice versa
   - Used with fork - parent process creates a pipe and uses it to communicate with a child process

Other IPC Mechanisms

3. Processes could communicate through variables that are shared between them
   - Shared variables, shared memory; other variables are private to a process
   - Special OS support for program to specify objects that are to be in shared regions of address space
   - Posix shared memory – shmget, shmat
4. Processes could communicate by sending and receiving messages to each other
   - Special OS support for these messages

More Ideas on IPC Mechanisms

5. Sometimes processes don't need to communicate explicit values to cooperate
   - They might just have to synchronize their activities
   - Example: Process 1 reads 2 matrices, Process 2 multiplies them, Process 3 writes the result matrix
   - Process 2 should not start work until Process 1 finishes reading, etc.
   - Called process synchronization
   - Synchronization primitives
     - Examples: mutex lock, semaphore, barrier

Programming With Shared Variables

- Consider a 2 process program in which both processes increment a shared variable

```
shared int X = 0;
P1:                  P2:
  X++;                 X++;
```

- *Q: What is the value of X after this?*
  - Want it to be X(P1) = 1 and X(P2) = 2, or vice versa
- But: X++ compiles into something like

```
LOAD  R1,    0(R2)
ADD   R1,    R1,   1
STORE 0(R2),       R1
```

Problem with shared variables

- Final value of X could be 1!

  P1 loads X into R1, increments R1

  ```
  LOAD  R1,   0(R2)
  ADD   R1,   R1,   1
  ```

  P2 loads X into register before P1 stores new value into X

  ```
  LOAD  R1,   0(R2)
  ADD   R1,   R1,   1
  ```

  Net result: P1 stores 1, P2 stores 1

  ```
  STORE 0(R2),      R1
  STORE 0(R2),      R1
  ```

- Race Condition – When result depends on exec order
- Need to synchronize 2 or more processes that try to update shared variable
- Critical Section: part of program where a shared variable is accessed

Critical Section Problem: Mutual Exclusion

- Must synchronize processes so that they access shared variable one at a time in critical section; called Mutual Exclusion (Mutex)
- Mutex Lock: a synchronization primitive
  - `AcquireLock(L)`
    - Done before critical section of code
    - Returns when safe for process to enter critical section
  - `ReleaseLock(L)`
    - Done after critical section
    - Allows another process to acquire lock

Implementing a Lock

```
int L=0; /* 0:lock available */

AcquireLock(L):
    while (L==1); /* BUSY WAITING */
    L = 1;

ReleaseLock(L):
    L = 0;
```

Why this implementation fails

```
while (L == 1);
L = 1;
```

```
wait: LW    R1, Addr(L)
      BNEZ  R1, wait
      ADDI  R1, R0, 1
      SW    R1, Addr(L)
```

Process 1	Process 2
LW R1 with 0	
Context Switch	
	LW R1 with 0
	BNEZ
	ADDI
	SW
	Enter CS
	Context Switch
BNEZ	
ADDI	
SW	
Enter CS	

time →

Assume that lock L is currently available (L = 0) and that 2 processes, P1 and P2 try to acquire the lock L

IMPLEMENTATION ALLOWS PROCESSES P1 and P2 TO BE IN CRITICAL SECTION TOGETHER!

Busy Wait Lock Implementation

- Hardware support will be useful to implement a lock
- Example: Test&Set instruction

```
Test&Set Lock:
    tmp = Lock
    Lock = 1
    Return tmp
```

Where these 3 steps happen atomically or indivisibly.

i.e., all 3 happen as one operation (with nothing happening in between)

Atomic Read-Modify-Write (RMW) instruction

Busy Wait Lock with Test&Set

```
AcquireLock(L)
    while (Test&Set(L)) ;
ReleaseLock(L)
    L = 0;
```

- Consider the case where P1 is currently in a critical section, P2-P10 are executing **AcquireLock**: all are executing the while loop
- When P1 releases the lock, by the definition of **Test&Set** exactly one of P2-P10 will read the new lock value of **0** and set L back to **1**
    - Other processes will continue to read this new value of 1

More on Locks

- Other names for this kind of lock
  - Mutex
  - Spin wait lock
  - Busy wait lock
- Can have locks where instead of busy waiting, an unsuccessful process gets blocked by the operating system

Semaphore

- A more general synchronization mechanism
- Operations: ***P*** (wait) and ***V*** (signal)
- ***P***(S)
  - if S is nonzero, decrements S and returns
  - Else, suspends the process until S becomes nonzero, when the process is restarted
  - After restarting, decrements S and returns
- ***V***(S)
  - Increments S by 1
  - If there are processes blocked for S, restarts exactly one of them

Critical Section Problem & Semaphore

- Semaphore S = 1;
- Before critical section: ***P***(S)
- After critical section: ***V***(S)
- Semaphores can do more than mutex locks
  - Initialize S to 10 and 10 processes will be allowed to proceed
  - P1:read matrices; P2: multiply; P3: write product
  - Semaphores S1=S2=0;
  - End of P1: ***V***(S1), beginning of P2: ***P***(S1) etc

Deadlock

Consider the following process:

P1: lock (L); lock(L);

- P1 is waiting for something (release of lock that it is holding) that will never happen
- Simple case of a general problem called deadlock
- Cycle of processes waiting for resources held by others while holding resources needed by others

Classical Problems

Producers-Consumers Problem

- Bounded buffer problem
- Producer process makes things and puts them into a fixed size shared buffer
- Consumer process takes things out of shared buffer and uses them

➢ Must ensure that producer doesn't put into full buffer or consumer take out of empty buffer

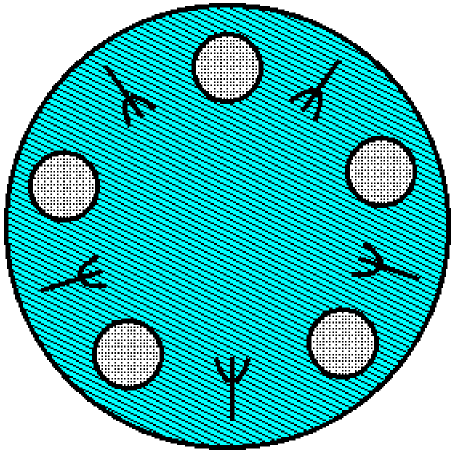
➢ While treating buffer accesses as critical section

Producers-Consumers Problem

```
shared Buffer[0 .. N-1]
Producer: repeatedly
   Produce x ; if (buffer full) wait for consumption
   Buffer[i++] = x ; signal consumer
Consumer: repeatedly
   If (buffer empty) wait for production;
   y = Buffer[- - i]
   Consume y ; signal producer
```

Dining Philosophers Problem

- *N* philosophers sitting around a circular table with a plate of food in front of each and a fork between each 2 philosophers
- Philosopher does: Repeatedly
  - *Eat (using 2 forks)*
  - *Think*
- Problem:
  - *Avoid deadlock*
  - *Be fair*



```
#define N 5 /* Number of philosphers */
#define RIGHT(i) (((i)+1) %N)
#define LEFT(i) (((i)==N) ? 0 : (i)+1)
typedef enum { THINKING, HUNGRY, EATING }
phil_state;
phil_state state[N];
semaphore mutex =1;
semaphore s[N]; /* one per philosopher, all 0 */

void philosopher(int process) {
  while(1) {
    think();
    get_forks(process);
    eat();
    put_forks(process);
  }
}
```

www.isi.edu/~faber/cs402/notes/lecture8.pdf

```
void test(int i) {
  if ( state[i] == HUNGRY &&
   state[LEFT(i)] != EATING &&
   state[RIGHT(i)] != EATING ){
    state[i] = EATING; V(s[i]);
  }
}
```

```
void get_forks(int i) {
  P(mutex);
  state[i] = HUNGRY;
  test(i);
  V(mutex);
  P(s[i]);
}
```

```
void put_forks(int i) {
  P(mutex);
  state[i]= THINKING;
  test(LEFT(i));
  test(RIGHT(i));
  V(mutex);
}
```

www.isi.edu/~faber/cs402/notes/lecture8.pdf

THREADS

Thread

- The basic unit of CPU utilization
- Thread of control in a process
- `Light weight process' (LWP)

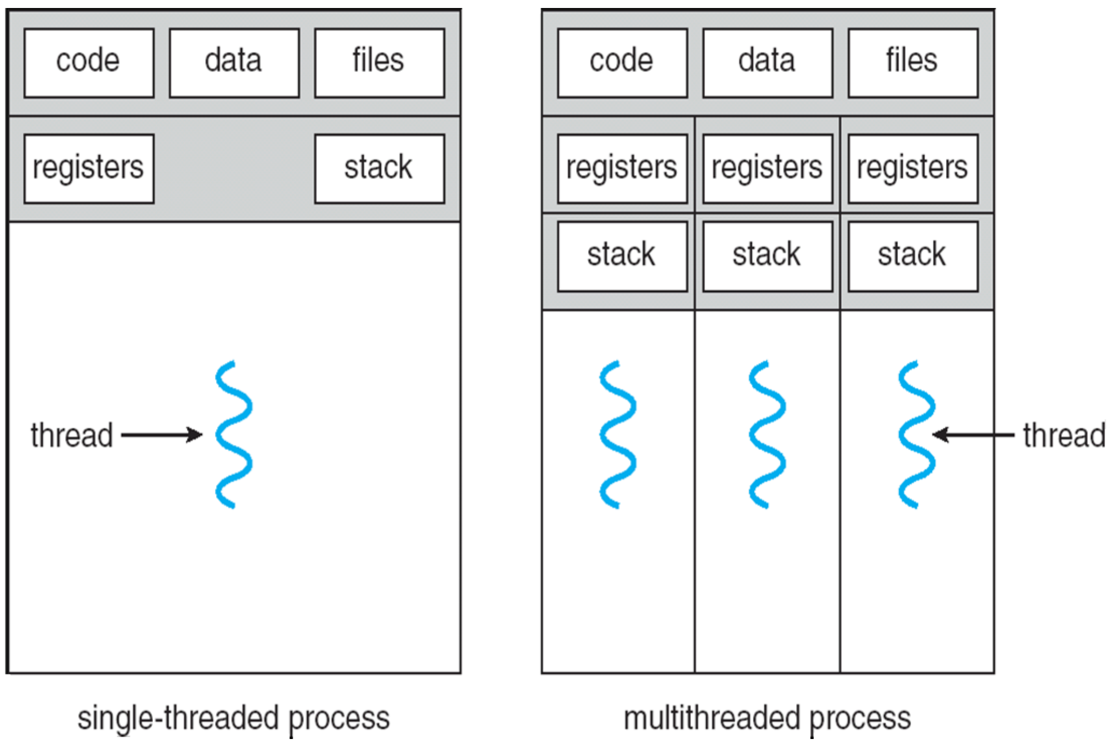
- *Weight* related to
  - Time for creation *(e.g. 30x faster than Process)*
  - Time for context switch *(e.g. 5x faster)*
  - Size of context
- Recall context of process

Threads and Processes

- Thread context
  - *Thread id, Stack, Stack Pointer, PC, Registers*
- So, thread context switching can be fast
- Many threads in same process that share parts of process context
  - Virtual address space (other than stack)
- So, *threads in the same process share variables* that are not stack allocated
- User (process) can manage synchronization of threads

Threads and Sharing

- Shares with other threads of a process – code section, data section, open files and signals



Threads

- Benefits – responsiveness, communication, parallelism and scalability
- Types – user threads and kernel threads
- Multithreading models
  - Many-one: efficient; but entire process will block if a thread makes a blocking system call
  - One-to-one: e.g. Linux. Parallelism; but can be heavy weight
  - Many-to-many: balance between the above two schemes

Thread Implementation

- Can be supported in the OS or by a library
- **Pthreads**: POSIX thread library – a standard for defining thread creation and synchronization
  - `int` **`pthread_create`**
    - `pthread_t *`*`thread`*`, const pthread_attr_t *`*`attr`*`, void *(*`*`start_routine`*`), void *`*`arg`*
  - `pthread_attr`
  - `pthread_join`
  - `pthread_exit`
  - `pthread_detach`
- Do "man –k pthreads"

See Sec 12.3-12.5, Bryant

code/conc/hello.c

```c
#include "csapp.h"
void *thread(void *vargp);

int main()
{
    pthread_t tid;
    Pthread_create(&tid, NULL, thread, NULL);
    Pthread_join(tid, NULL);
    exit(0);
}

void *thread(void *vargp) /* Thread routine */
{
    printf("Hello, world!\n");
    return NULL;
}
```

code/conc/hello.c

Figure 12.13 **hello.c: The Pthreads "Hello, world!" program.**

Sec 12.3, Bryant

Synchronization Primitives

Mutex locks

`int pthread_mutex_lock(pthread_mutex_t *mutex)`

If the mutex is already locked, the calling thread blocks until the mutex becomes available. Returns with the mutex object referenced by mutex in the locked state with the calling thread as its owner.

`pthread_mutex_trylock`

`pthread_mutex_unlock`

Semaphores

```
sem_init
sem_wait
sem_post
```

Pthread scheduling

- *Process contention scope* – scheduling user-level threads among a set of kernel threads.
- *System contention scope* – scheduling kernel threads for CPU.
- Functions for setting the scope
  - `pthread_attr_setscope`,
  - `pthread_attr_getscope`
  - Can use `PTHREAD_SCOPE_PROCESS` for PCS and `PTHREAD_SCOPE_SYSTEM` for SCS

Thread Safety

- A function is thread safe if it always produces correct results when called repeatedly from concurrent multiple threads
- Thread Unsafe functions
  - That don't protect shared variables
  - That keep state across multiple invocations
  - That return a pointer to a static variable
  - That call thread unsafe functions
- Races
  - When correctness of a program depends on one thread reaching a point x before another thread reaching a point y

Recommended Reading

- Silberschatz
  - Chapter 6: Process Synchronization
  - Chapter 4: Threads
- Bryant: Chapter 12.3—12.5, Concurrent Threads

Schedule

- Wed 22 Oct, 8AM: Parallelization
- Tue, Thu 28 & 30 Oct: Parallel Architectures (RG)
- Substitute Classes: Sat 1 Nov, Wed 5 Nov, Sat 8 Nov: MPI