CIS 5512 - Operating Systems
IPC and Synchronization

Professor Qiang Zeng
Fall 2015
Previous class...

- Process state transition
  - Ready, blocked, running

- Context switch
  - Process switch
  - Interrupt/exception handling

- How to locate kernel stack and PCB in Linux

- Interrupt vs. exception vs. signal

- Calling convention

- `fork()`/`wait()` and Shell
Outline

• IPC
• Concepts in concurrent computing
• Synchronization primitives
• Synchronization primitives in Linux userspace
IPC - Pipes

Example: `ls | sort`

```c
FILE *pipe_fp, *infile;
char readbuf[80];

/* Open up input file */
infile = fopen(fileName, "rt");
/* Create one way pipe line with call to popen() */
  It does pipe creation, fork, and exec
*/
pipe_fp = popen(childProgramName, "w");
while(true) {
    fgets(readbuf, 80, infile);
    if(feof(infile)) break;
    fputs(readbuf, pipe_fp);
}

fclose(infile);
pclose(pipe_fp); // close the pipe and wait()
```

parent
pipe_fp

STDIN_FILENO = 0

child

read() a closed pipe will return error!
```c
#define FIFO_FILE "MYFIFO"

FILE *fp;
char readbuf[80];

/* Create the FIFO if it does not exist */
umask(0);
mknod(FIFO_FILE, S_IFIFO|0666, 0);

fp = fopen(FIFO_FILE, "r");
fgets(readbuf, 80, fp);
printf("Received string: %s\n", readbuf);
fclose(fp);
```

```c
#define FIFO_FILE "MYFIFO"

fp = fopen(FIFO_FILE, "w");
fputs("Hello!", fp);
fclose(fp);
return(0);
```
# IPC – Message queues

Essentially, a linked list in kernel

<table>
<thead>
<tr>
<th>API</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>keyval = ftok(&quot;.&quot;, 'm');</td>
<td>Generate a key described by “.&quot;&quot;, “m”</td>
</tr>
<tr>
<td>qid = msgget( keyval, IPC_CREAT</td>
<td>0660 )</td>
</tr>
<tr>
<td>msgsnd( qid, qbuf, length, 0)</td>
<td>Sender</td>
</tr>
<tr>
<td>msgrcv( qid, NULL, 0, type, IPC_NOWAIT)</td>
<td>Receiver</td>
</tr>
</tbody>
</table>
## Compare different IPCs

<table>
<thead>
<tr>
<th>IPC method</th>
<th>Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pipes</td>
<td>Can only be used among parent and child</td>
</tr>
<tr>
<td>FIFO (named pipes)</td>
<td>Can be referred to by a string, so doesn’t have the limitation above</td>
</tr>
<tr>
<td>Message queues</td>
<td>Supports message boundary and message types</td>
</tr>
<tr>
<td>Shared memory</td>
<td>Data passing doesn’t go through kernel, so it is usually the most efficient one</td>
</tr>
</tbody>
</table>
Race condition bug

- A race condition exists when the final program result depends on the execution sequence.
- Let’s consider a counter, c, that is incremented by multiple processes concurrently.

```assembly
++c;
(1) movl c, %eax
(2) addl $1, %eax
(3) movl %eax, c
```

- Now, assume c = 0 initially, and two processes increment it.
- Consider the execution sequence: after process 0 executes (1) (2), it is scheduled out; and then process 1 executes (1) – (3)?
- You get c=1 here, while in other execution sequences you may get 2
- A concurrent program as simple as this has a race condition bug.
Intuitive attempts to fix the bug

Intuitive attempts don’t work; we need a formal treatment by introducing new concepts

- Use atomic_add instructions
  - atomic_add supports you to do atomic increment
  - It works in that extremely simple example
  - But it is not a general solution; consider another example

```c
// critical section: withdraw $100 from account
(1) if(account >= 100)
(2)    account -= 100;
// Is it possible to withdraw $200, given account = 100?
```

- Disable interrupts to prevent the current process from being schedule out
  - This cannot prevent multiple processes on a multi-core from withdrawing money concurrently and happily
Critical section and mutual exclusion

– the paper started the era of concurrent computing

• A **Critical section** is a program region that accesses shared data
• **Mutual exclusion** is to make sure no two processes are simultaneously inside their critical sections
• A good solution to mutual exclusion should satisfy
  – Mutual exclusion
  – No assumptions made about processor speed or number
  – No outside blocker
  – No infinite wait
• Such a solution is based on synchronization primitives

```c
enter_region();
// Critical section;
leave_region();
```
A failed attempt: strict alternation

What if process 0 quickly finishes one iteration and wants to execute another, while process 1 has not entered the critical section?

It violates Condition 3 “no outside blocker”
Dekker’s algorithm - First correct solution

Pointed out by Dijkstra in “Cooperating Sequential Processes”, 1965 Sept, which also proposed Semaphore.

```c
//flag[] is boolean array; and turn is an integer
flag[0] = false
flag[1] = false
turn = 0  // or 1

P0:
flag[0] = true;
while (flag[1] == true) {
  if (turn != 0) {
    flag[0] = false;
    while (turn != 0) {
      // busy wait
    }
    flag[0] = true;
  }
}

// critical section
...
turn = 1;
flag[0] = false;
// remainder section

P1:
flag[1] = true;
while (flag[0] == true) {
  if (turn != 1) {
    flag[1] = false;
    while (turn != 1) {
      // busy wait
    }
    flag[1] = true;
  }
}

// critical section
...
turn = 0;
flag[1] = false;
// remainder section
```
Compiler optimization and memory ordering

• Issues due to compiler optimizations
  – In the view of Process 0, write to flag[0] is wasted, flag[1] should be cached in registers, and turn has no hope to be changed, so they are optimized by compiler
  – “volatile” ensures that access to volatile variables are not removed or cached

• Issues due to memory ordering, e.g., between loads and stores
  – memory fences are needed
  – GCC: __sync_synchronize() // both a compiler and h/w barrier
  – asm volatile("" ::: "memory") is insufficient, as it is just a compiler barrier

```
x = 3;
y = 4;
x = 5;
while(x == 5) i++;
gcc -O2
```

```
x = 5;
y = 4;
while(true) {
  // critical section
  ...
  turn = 1;
  flag[0] = false;
  // remainder section
}
```

```
P0:
  flag[0] = true;
  while (flag[1] -- true) {
    if (turn ≠ 0) {
      flag[0] = false;
      while (turn ≠ 0) {
        // busy wait
      }
      flag[0] = true;
    }
  }
  flag[0] = true;

P1:
  flag[1] = true;
  while (flag[0] -- true) {
    if (turn ≠ 1) {
      flag[1] = false;
      while (turn ≠ 1) {
        // busy wait
      }
      flag[1] = true;
    }
  }
  flag[1] = true;

// critical section
...
Peterson’s algorithm – simplify Dekker’s

```c
#define FALSE  0
#define TRUE   1
#define N      2    /* number of processes */

int turn;        /* whose turn is it? */
int interested[N];/* all values initially 0 (FALSE) */

void enter_region(int process);    /* process is 0 or 1 */
{
    int other;    /* number of the other process */

    other = 1 - process;    /* the opposite of process */
    interested[process] = TRUE; /* show that you are interested */
    turn = process;        /* set flag */
    while (turn == process && interested[other] == TRUE) /* null statement */ ;
}

What if two processes execute “turn = process” almost the same time?

void leave_region(int process)    /* process: who is leaving */
{
    interested[process] = FALSE;    /* indicate departure from critical region */
}
```

What if two processes execute “turn = process” almost the same time?
Lamport’s bakery algorithm – n processes

“A New Solution of Dijkstra's Concurrent Programming Problem”, Lamport 1974

```c
lock (int pid) {
    // enter doorway
    choosing[pid] = 1;
    number[pid] = 1+max(number[0],...,number[N-1]);
    choosing[pid] =0;

    // enter bakery
    while(j<N) {
        if (j != pid) {
            while(choosing[j]); // spin
            while(number[j] &&
                ((number[j] < number[pid]) ||
                ((number[j]==number[pid]) && j<pid))
            ); // spin
            j++;
        }
    }
}

unlock (int pid) { number[pid] = 0; }
```

Process j’s number is not settled, so wait
If (j < pid), let j win
Solutions based on atomic read-modify-write instructions

test_and_set(int* p){
    int t = *p;
    *p = 1;
    return t;
}

//compare-and-swap
CAS(p, old, nvalue){
    if(*p != old)
        return false;
    *p = nvalue;
    return false;
}

Load-link/store-conditional, or LL/SC // CAS equivalent in RISC architectures

XCHG(int *x, int *p){
    int t = *x;
    *x = *p;
    *p = t;
}

enter_region() {
    while(test_and_set(&lock) == 1);
}

leave_region() { lock = 0;}

enter_region() {
    a = 1;
    do { XCHG(&a, &lock);
        } while ( a == 1)
}

leave_region() { lock = 0;}
A faulty solution to the Producer-consumer problem

```c
int itemCount = 0;

procedure producer() {
    while (true) {
        item = produceItem();
        if (itemCount == BUFFER_SIZE) {
            sleep();
        }
        putItemIntoBuffer(item);
        itemCount = itemCount + 1;
        if (itemCount == 1) {
            wakeup(consumer);
        }
    }
}

procedure consumer() {
    while (true) {
        item = removeItemFromBuffer();
        itemCount = itemCount - 1;
        if (itemCount == BUFFER_SIZE - 1) {
            wakeup(producer);
        }
        consumeItem(item);
    }
}
```

Wakeup messages may be lost, when itemCount changes from 0 to 1, or from BUFFER_SIZE to BUFFER_SIZE-1
Semaphore – avoids busy-waiting

- Solutions so far perform busy waiting
- Semaphore
  - Contains an integer to indicate the number of resources
  - Contains a queue for processes waiting for the semaphore
  - The process sleeps (instead of busy waiting) if the number is 0

```
// Atomic
down(S) { // or, P operation
    if(S.num>0)
        S.num--;
    else
        put the current process in queue;
    block the current process;
}
```

```
// Atomic
up(S) { // or, V operation
    if(any process is in S’s wait queue)
        pop a process from the queue;
        resume it;
    else
        S.num++;
}
```
Semaphore-based solution to the multi-producer multi-consumer problem

- Two semaphore (`fillCount` and `emptyCount`) are used to sleep/awaken waiting producers and consumers
- A binary semaphore (`S.num` can only be 0 or 1) can be used as a mutex (mutual exclusion); recall `enter_region()` and `exit_region()`

```plaintext
semaphore mutex = 1;
semaphore fillCount = 0;
semaphore emptyCount = BUFFER_SIZE;

procedure producer() {
  while (true) {
    item = produceItem();
    down(emptyCount);
    down(mutex);
    putItemIntoBuffer(item);
    up(mutex);
    up(fillCount);
  }
}

procedure consumer() {
  while (true) {
    down(fillCount);
    down(mutex);
    item = removeItemFromBuffer();
    up(mutex);
    up(emptyCount);
    consumeItem(item);
  }
}
```

Is mutex required, if there is only one producer and one consumer?
Mutex vs. Semaphore – my email

- Mutex is to indicate whether a privilege is being occupied or not. For example, the privilege of incrementing a counter, and the privilege to accessing "account"
- Semaphore is to indicate the number of resources. For example, in the producer-consumer problem, in the view of a producer, an empty slot is a piece of resource, which is waited for by a producer but generated by a consumer.
- A binary semaphore (whose internal counter only be either 0 or 1) can be used as a mutex, but that is just a special use of semaphore. You regard the privilege as one piece of resource
- All the mutexes we covered use busy-waiting (so they are called spinlocks), while semaphore turns a process to sleep (or, be blocked), so it saves CPU cycles in some sense (why semaphore doesn't always save CPU time?)
# Semaphore and pthread_mutex in Linux

<table>
<thead>
<tr>
<th>function</th>
<th>explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sem_open()</code></td>
<td>Create or connect to a named semaphore (and increment the reference count), which can be conveniently shared inter-process</td>
</tr>
<tr>
<td><code>sem_unlink()</code></td>
<td>Remove the named semaphore once its reference count = 0</td>
</tr>
<tr>
<td><code>sem_close()</code></td>
<td>Decrement the reference count (<code>exit()</code> does this automatically)</td>
</tr>
<tr>
<td><code>sem_init()</code></td>
<td>Create an unnamed semaphore (usually a global variable)</td>
</tr>
<tr>
<td><code>sem_destroy()</code></td>
<td>Destroy an unnamed semaphore</td>
</tr>
<tr>
<td><code>sem_wait()</code></td>
<td>P operation</td>
</tr>
<tr>
<td><code>sem_post()</code></td>
<td>V operation</td>
</tr>
<tr>
<td><code>pthread_mutex_init()</code></td>
<td>PTHREAD_PROCESS_SHARED: process-shared mutex Other APIs: <code>*_destroy()</code>, <code>*_lock()</code>, <code>*_trylock()</code>, <code>*_unlock()</code></td>
</tr>
</tbody>
</table>
Conditional variables (CV)

• If we revisit the previous faulty solution to the producer-consumer problem, we can notice it is natural and easy to code, while the solution based on semaphore is subtle.

• Is there an easy and correct solution?

<table>
<thead>
<tr>
<th>Function</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>pthread_cond_wait (cond, mutex)</td>
<td><strong>Precondition:</strong> the calling process must own the mutex before calling this function.</td>
</tr>
<tr>
<td></td>
<td>It atomically releases the mutex and waits on a condition variable.</td>
</tr>
<tr>
<td></td>
<td>When it is unblocked by <code>pthread_cond_signal(cond)</code>, the calling process contends for the mutex as if as if it had called <code>pthread_mutex_lock()</code></td>
</tr>
</tbody>
</table>
A solution based on mutex and CV

typedef struct {
    char buf[BSIZE];
    int occupied, in, out;
    pthread_mutex_t mutex;
    pthread_cond_t slot, item;
} buffer_t;
buffer_t b;

void producer(char c)
{
    pthread_mutex_lock(&b->mutex);
    while (b->occupied >= BSIZE)
        pthread_cond_wait(&b->slot, &b->mutex);
    b->buf[b->in] = c;
    b->in = (b->in+1) % BSIZE;
    b->occupied++;
    pthread_cond_signal(&b->item);
    pthread_mutex_unlock(&b->mutex);
}

char consumer()
{
    char c;
    pthread_mutex_lock(&b->mutex);
    while (b->occupied <= 0)
        pthread_cond_wait(&b->item, &b->mutex);
    i = b->buf[b->out];
    b->out %= (b->out + 1) % BSIZE;
    b->occupied--;
    pthread_cond_signal(&b->slot);
    pthread_mutex_unlock(&b->mutex);
    return(c);
}

This is much easier to code and understand!
Can we do even better? We’ll see.
Monitor

- A synchronization construct that encapsulates mutex and conditional variable

<table>
<thead>
<tr>
<th>W/o monitor</th>
<th>With monitor (e.g., in Java)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A procedure manually acquires and releases the mutex</td>
<td>Each synchronized procedure does so automatically (enforced by the compiler)</td>
</tr>
<tr>
<td>pthread_cond_wait (cond, mutex)</td>
<td>wait()</td>
</tr>
<tr>
<td>pthread_cond_signal(cond)</td>
<td>notify()</td>
</tr>
<tr>
<td>pthread_cond_broadcast(cond)</td>
<td>notifyAll()</td>
</tr>
</tbody>
</table>
Readers and writers problem

Reads are writes are mutual exclusive
Concurrent reads are allowed; concurrent writes are not.

```c
// mutex for rc
semaphore rc_mutex = 1;
// mutex for db
semaphore db = 1;

void writer(void) {
    while(true) {
        d = get_data();
        down(&db);
        write_data(d);
        up(&db)
    }
}

void reader(void) {
    while(true) {
        down(&rc_mutex);
        ++rc;
        if(rc == 1)  down(&db);
        up(&rc_mutex);
        d = read_data();
        down(&rc_mutex);
        --rc;
        if(rc == 0)  up(&db);
        up(&rc_mutex);
    }
}
```
Issues with locks (mutex and semaphore)

• You may introduce deadlock
• Nobody can make progress, when the lock owner is scheduled out
• Priority inversion: a high priority is waiting for the lock, while it is owned by a low priority process
A lock-free solution to single-producer single-consumer problem

```c
volatile unsigned int produceCount, consumeCount;
char buffer[BUFFER_SIZE];

void producer(void) {
    while (1) {
        while (produceCount - consumeCount == BUFFER_SIZE)
            sched_yield(); // buffer is full

        buffer[produceCount % BUFFER_SIZE] = produceChar();
        ++produceCount;
    }
}

void consumer(void) {
    while (1) {
        while (produceCount - consumeCount == 0)
            sched_yield(); // buffer is empty

        consumeChar(buffer[consumeCount % BUFFER_SIZE]);
        ++consumeCount;
    }
}
```

The lock still has its advantage: it can be used to address general concurrent computing problems, while there are only limited types of lock-free data structures.
Big picture of synchronization primitives

Spin locks
(busy-waiting locks)

- Algorithms that do not rely on special instructions (Dekker’, Peterson’s, Bakery)
- Algorithms based on atomic read-modify-write instructions (test-and-set, xchg)

Mutex: like Spin locks, it also has the lock()/unlock() APIs; the only difference is that the calling thread blocks instead of busy-waiting

Semaphore: it contains an internal counter indicating the number of resources.

Binary Semaphore is a special semaphore, whose counter value can only be 0 or 1; sometimes it is used as a mutex

Conditional Variables: they are used to ease concurrent programming. You have to use it with a mutex. Monitor encapsulates CV and mutex

Sync. Primitives (or loosely, “locks”)

Blocking locks
Take away…

• IPC
  – Pipes, FIFO (named pipes), message queues, and shared memory
• Concepts
  – Race condition, critical section, mutual exclusion
• Pure software based locks
  – Dekker’s, Peterson’s, Bakery algorithms
• Atomic read-modify-write (RMW) instructions
  – Test-and-set, xchg, compare-and-swap
  – Busy-waiting lock based on (RMW)
• Semaphores and binary semaphores
• Conditional variables
• Monitors
• Classic concurrency problems
  – Producer-consumer, reader-writer
• Lock-free concurrent computing