Review: Summary of Lecture 1

° Two major OS functionalities:
  • machine extension and resource management

° History of OS:
  • Batching, multiprogramming, time-sharing, PC

° Computer Architecture Reviews

° Fundamental OS concepts:
  • Process, memory management, deadlocks, file & directory management

° System calls and Unix Shell

° More reading: textbook 1.1 - 1.7, Ch10: 671 - 696
Chapter 2
Processes and Threads

2.1 Processes
2.2 Threads
2.3 Interprocess communication
2.4 Classical IPC problems
2.5 Scheduling

Process
° A fundamental OS abstraction
  • A program in execution
  • An instance of a program running on a compute
  • The entity that can be assigned to and executed on a processor
  • A unit of activity characterized by the execution of a sequence of instructions, a current state, and an associated set of system instructions
The Process Model

- (a) Multiprogramming of four programs
- (b) Conceptual model of 4 independent, sequential processes
  - Sequential process mode: hiding the effects of interrupts
- (c) Only one program active at any instant

What is pseudo-parallelism?

What is the key difference between a program and a process?

Process Creation

Principal events that cause process creation

1. System initialization; foreground and background
2. Execution of a process creation system
3. User request to create a new process; interactive systems
4. Initiation of a batch job

Technically, a new process is created by having an existing process execute a process creation system call

UNIX: a two-step creation-loading process of fork() and execve()

WINDOWS: a single Win32 call CreateProcess()
Process Termination

Conditions which terminate processes
1. Normal exit (voluntary)
2. Error exit (voluntary)
3. Fatal error (involuntary)
4. Killed by another process (involuntary)
   UNIX: kill()
   Windows32: TerminateProcess()

Process Hierarchies

° Parent creates a child process, child processes can create its own process
° Forms a hierarchy
  • UNIX calls this a "process group"
  • init, a special process is present in the boot image
° Windows has no concept of process hierarchy
  • all processes are created equal
Possible process states (a three-state process model)
- running
- blocked
- ready

Transitions between states shown:
1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available

Lowest layer of process-structured OS
- handles interrupts, scheduling

Above that layer are sequential processes
Using Two Queues

- When an event occurs, the OS must scan the entire blocked queue, searching for those processes waiting on that event.
- In a large OS, there could be hundreds or thousands of processes in that queue.

Multiple Blocked Queues

- (b) Multiple blocked queues
Implementation of Processes

Fields of a process table entry

Process Execution

Program Counter

0000

Main Memory

0

100

Dispatcher

500

Process A

800

Process B

1300

Process C
### When to Switch a Process

- **Clock interrupt**
  - process has executed for the maximum allowable time slice
- **I/O interrupt**
- **Memory fault**
  - memory address is in virtual memory so it must be brought into main memory
- **Trap**
  - error occurred
  - may cause process to be moved to Exit state
- **Supervisor call**
  - such as file open

### OS' Interrupt Handling

- **Interrupt vector**
  - contains the address of the interrupt service procedures

1. Hardware stacks program counter, etc.
2. Hardware loads new program counter from interrupt vector.
3. Assembly language procedure saves registers.
4. Assembly language procedure sets up new stack.
5. C interrupt service runs (typically reads and buffers input).
6. Scheduler decides which process is to run next.
7. C procedure returns to the assembly code.
8. Assembly language procedure starts up new current process.

Skeleton of what lowest level of OS does when an interrupt occurs when a process is running
Operating System Control Structures

- Information about the current status of each process and resource
- Tables are constructed for each entity the operating system manages
- Four typical types of tables:
  - Memory tables
  - I/O tables
  - File tables
  - Process tables

Structure of OS Control Tables

Figure: General Structure of Operating System Control Tables
Concurrency and Parallelism

- Concurrency in software is a way to manage the sharing of resources efficiently at the same time
  - When multiple software threads of execution are running concurrently, the execution of the threads is interleaved onto a single hardware resource
    - Why not schedule another thread while one thread is on a cache miss or even page fault?

- True parallelism requires multiple hardware resources
  - Multiple software threads are running simultaneously on different hardware resources/processing elements

Thread and Multithreading

- Process: for resource grouping and execution
- Thread: a finer-granularity entity for execution and parallelism
  - Lightweight processes, multithreading
- Multi-threading: Operating system supports multiple threads of execution within a single process
- UNIX supports multiple user processes but only supports one thread per process
- Windows 2000, Solaris, Linux, Mach, and OS/2 support multiple threads
**The Thread Model**

- Can we use model (a) for a file server using a cache in memory? Then why not each process has its own cache? How about model (b)?

(a) Three processes each with one thread, but different address spaces  
(b) One process with three threads, sharing the address space

**The Thread Model (2)**

- What is a key reason to have multi-threads in a process?  
  - Resource sharing for higher (pseudo-)parallelism  
  - But timesharing threads is no different than timesharing processes from scheduling perspective

<table>
<thead>
<tr>
<th>Per process items</th>
<th>Per thread items</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address space</td>
<td>Program counter</td>
</tr>
<tr>
<td>Global variables</td>
<td>Registers</td>
</tr>
<tr>
<td>Open files</td>
<td>Stack</td>
</tr>
<tr>
<td>Child processes</td>
<td>State</td>
</tr>
<tr>
<td>Pending alarms</td>
<td>Scheduling properties</td>
</tr>
<tr>
<td>Signals and signal handlers</td>
<td></td>
</tr>
<tr>
<td>Accounting information</td>
<td></td>
</tr>
</tbody>
</table>

Items shared by all threads in a process  
Items private to each thread

Is there protection between threads? Why?
The Thread Model (3)

- What are private to a thread?
  - Stack for local procedures' local variables and the return address
  - Why a thread needs its own stack?
  - Library procedures: `thread_create()`, `thread_exit()`, `thread_yield()`

![Thread Model Diagram]

Timesharing between threads is a job of programmers! Why?

Why Multi-threading

- Performance!
  - Lightweight (less overhead to create, terminate, and switch)
  - Overlapping CPU work with I/O
  - Priority/real-time scheduling
  - Thread communication in a process without trap to kernel

<table>
<thead>
<tr>
<th>Platform</th>
<th><code>fork()</code></th>
<th><code>pthread_create()</code></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>real</td>
<td>user</td>
</tr>
<tr>
<td>IBM 375 MHz POWER3</td>
<td>61.94</td>
<td>3.49</td>
</tr>
<tr>
<td>IBM 1.5 GHz POWER4</td>
<td>44.08</td>
<td>2.21</td>
</tr>
<tr>
<td>IBM 1.9 GHz POWER5 p5-575</td>
<td>50.66</td>
<td>3.32</td>
</tr>
<tr>
<td>INTEL 2.4 GHz Xeon</td>
<td>23.81</td>
<td>3.12</td>
</tr>
<tr>
<td>INTEL 1.4 GHz Titanium 2</td>
<td>23.61</td>
<td>0.12</td>
</tr>
</tbody>
</table>

Time results on SMP systems
Thread Usage – Finer-granularity for Parallelism

° Why multi-threading?
   • The ability for the parallel entities to share an address space and all of its data
   • Lightweight: easy to create and destroy, since no resource attached
   • Allow overlapping of I/O and CPU in a process, finer-granularity for (pseudo-)parallelism

Why three separate Processes do not work?

User interaction
Document reformatting
Disk backup
Disk


Consider reading a file using a single-threaded file server and a multi-threaded file server. It takes 10 ms to get a request for work, dispatch it, and do the rest of work, assuming the data needed is in the cache. If a disk operation is needed, as is the case 25% of the time, an additional 40 ms is required, during which time the thread sleep.

What is the throughput (# requests/sec) can the server handle, if it is single-threaded? If it is multi-threaded (assuming non-blocking file reading in the disk)?
Thread Usage – A Multi-threaded Web Server

° A dispatcher thread and worker threads
  • Shared Web page cache
  • Asynchronous event handling: working on previous requests and managing the arrival of new requests

---

Rough outline of code for previous slide
(a) Dispatcher thread
(b) Worker thread

```
while (TRUE) {
    get_next_request(&buf);
    handoff_work(&buf);
}

while (TRUE) {
    wait_for_work(&buf)
    look_for_page_in_cache(&buf, &page);
    if (page_not_in_cache(&page))
        read_page_from_disk(&buf, &page);
    return_page(&page);
}
```

What should be included in `handoff_work()` and `return_page()` for thread time-sharing?
Thread Usage – Alternatives to Multi-threading

- A multi-process but single-threaded Web server
  - Lower throughput
- Finite-state machine: a multi-process single-threaded Web server w/ non-blocking system calls
  - The “sequential process model” is lost
  - Need non-blocking system calls

<table>
<thead>
<tr>
<th>Model</th>
<th>Characteristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Threads</td>
<td>Parallelism, blocking system calls</td>
</tr>
<tr>
<td>Single-threaded process</td>
<td>No parallelism, blocking system calls</td>
</tr>
<tr>
<td>Finite-state machine</td>
<td>Parallelism, nonblocking system calls, interrupts</td>
</tr>
</tbody>
</table>

Three ways to construct a Web server

Implementing Threads in User Space

- User-space threads: the kernel knows nothing about them
  - No clock interrupts, no preemption

A user-level threads package
Implementing Threads in User Space - Discussions

° Advantages
  • No OS thread-support need
  • Lightweight: thread switching vs. process switching
    - Local procedure vs. system call (trap to kernel)
    - When we say a thread come-to-life? SP & PC switched
  • Each process has its own customized scheduling algorithms
    - thread_yield()

° Disadvantages
  • How blocking system calls implemented? Called by a thread?
    - Goal: to allow each thread to use blocking calls, but to prevent one blocked thread from affecting the others
  • How to change blocking system calls to non-blocking?
  • Jacket/wrapper: code to help check in advance if a call will block
  • How to deal with page faults?
  • How to stop a thread from running forever? No clock interrupts

Implementing Threads in the Kernel

° kernel-space threads: when a thread blocks, kernel re-schedules
  • Thread re-cycling: thread creation and termination more costly

A threads package managed by the kernel
Hybrid Implementations

- Use kernel-level threads and then *multiplex* user-level threads onto some or all of the kernel-level threads
  - Example Solaris

Scheduler Activations (Self-Reading)

- Goal – mimic functionality of kernel threads
  - gain performance of user space threads
- Avoids unnecessary user/kernel transitions
- Kernel assigns virtual processors to each process
  - lets runtime system allocate threads to processors
- Problem:
  - Fundamental reliance on kernel (lower layer)
  - calling procedures in user space (higher layer)
Pop-Up Threads

- Dynamic creation of a new thread when a request/message arrives
  - Each one starts fresh and each one is identical to each other

(a) before message arrives                  (b) after message arrives

Making Single-Threaded Code Multithreaded (1)

- Issue: variables global to a thread but not to the entire program
  - How about to prohibit global variables in a process?
**Making Single-Threaded Code Multithreaded (2)**

- Solution: to assign each thread its own private global variables

```c
create_global("bufptr");
set_global("bufptr", &buf);
bufptr = read_global("bufptr");
```

Threads can have private global variables

---

**Hardware Threading**

- At hardware level, a thread is an execution path that remains independent of other hardware thread execution paths

```
User-level Threads
Used by applications and handled by user-level runtime

Kernel-level Threads
Used and handled by OS kernel

Hardware Threads
Used by each processor
```

Operational flow
Simultaneous Multi-threading (Hyper-Threading)

- Simultaneous multi-threading (SMT)
  - Create multiple logical processors with a physical processor
  - One logical processor for a thread, which requires an architecture state consisting of the GPRs and Interrupt logic
  - Duplicate multiple architecture states (CPU state), and, let other CPU resources, such as caches, buses, execution units, branch prediction logic shared among architecture states
  - Multi-threading at hardware level, instead of OS switching
  - Intel's SMT implementation is called Hyper-Threading Technology (HT Technology)

Multi-Processor and Multi-Core

- multi-core processors use chip multi-processing (CMP)
  - Cores are essentially two individual processors on a single die
  - May or may not share on-chip cache
  - True parallelism, instead of high concurrency
Inter-Process Communication (IPC)

- Three fundamental issues:
  - How one process can pass information to another
  - How to make sure two or more processes do not get into each other’s way when engaging in critical activities
  - How to maintain proper sequencing when dependencies present

IPC: Race Conditions

- Race conditions: when two or more processes/threads are reading or writing some shared data and the final results depends on who runs precisely when
  - Interrupts, interleaved operations/execution

Two processes want to access shared memory at same time
### IPC: Race Conditions (2)

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read balance; $1000</td>
<td>$1000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Read balance; $1000</td>
<td>$1000</td>
</tr>
<tr>
<td></td>
<td>Deposit $200</td>
<td>$1000</td>
</tr>
<tr>
<td>Deposit $200</td>
<td></td>
<td>$1000</td>
</tr>
<tr>
<td>Update balance $1000 + $200</td>
<td>$1200</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Update balance $1000 + $200</td>
<td>$1200</td>
</tr>
</tbody>
</table>

Two threads want to deposit an account; overwriting issue

### Mutual Exclusion and Critical Regions

- Mutual exclusion: makes sure if one process is using a shared variable or file, the other processes will be excluded from doing the same thing
  - Main challenge/issue to OS: to design appropriate primitive operations for achieving mutual exclusion

- Critical regions: the part of the program where the shared memory is accessed

- Four conditions to provide mutual exclusion
  - No two processes simultaneously in critical region
  - No assumptions made about speeds or numbers of CPUs
  - No process running outside its critical region may block another process
  - No process must wait forever to enter its critical region
Mutual Exclusion Using Critical Regions

Mutual Exclusion with Busy Waiting (1)

- Disabling interrupts: OS technique, not users’
- Lock variables: test-set is a two-step process, not atomic
- Busy waiting: continuously testing a variable until some value appears

```
while (TRUE) {
    while (turn != 0) /* loop */;
    critical_region();
    turn = 1;
    noncritical_region();
}
```

(b) Proposed strict alternation solution to critical region problem
(a) Process 0. (b) Process 1.
Mutual Exclusion with Busy Waiting (2) – Peterson’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 */;
}

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

Peterson's solution for achieving mutual exclusion
```

Mutual Exclusion with Busy Waiting (3) - TSL

- **TSL (Test and Set Lock)**
  - Indivisible (atomic) operation, how? Hardware (multi-processor)
  - How to use TSL to prevent two processes from simultaneously entering their critical regions?

```
enter_region:
    TSL REGISTER,LOCK | copy lock to register and set lock to 1
    CMP REGISTER,#0    | was lock zero?
    JNE enter_region   | if it was non zero, lock was set, so loop
    RET                 | return to caller; critical region entered

leave_region:
    MOVE LOCK,#0       | store a 0 in lock
    RET                 | return to caller
```

Entering and leaving a critical region using the TSL instruction

*(if one process cheats, the mutual exclusion will fail!)*
Sleep and Wakeup

° Issue I with Peterson’s & TS: how to avoid CPU-costly busy waiting?

° Issue II: priority inversion problem

- Consider two processes, H with (strict) high priority and L with (strict) low priority, L is in its critical region and H becomes ready; does L have chance to leave its critical region?
- Can this problem occur with user-space threads? Why?
- POSIX pthread_mutex_trylock() is non-blocking call

° Some IPC primitives that block instead of wasting CPU time when they are not allowed to enter their critical regions
  - Sleep and wakeup

Sleep and Wakeup – Producer-Consumer Problem

```c
#define N 100
int count = 0;

void producer(void)
{
    int item;
    while (TRUE) 
    {
        item = produce_item(); /* generate next item */
        if (count == N) sleep(); /* if buffer is full, go to sleep */
        insert_item(item); /* put item in buffer */
        count = count + 1; /* increment count of items in buffer */
        if (count == 1) wakeup(consumer); /* was buffer empty? */
    }
}

void consumer(void)
{
    int item;
    while (TRUE) 
    {
        if (count == 0) sleep(); /* if buffer is empty, go to sleep */
        item = remove_item(); /* take item out of buffer */
        count = count - 1; /* decrement count of items in buffer */
        if (count == N - 1) wakeup(producer); /* was buffer full? */
        consume_item(item); /* print item */
    }
}
```

Q: What if the wakeup signal sent to a non-sleep (ready) process?

Producer-consumer problem with fatal race condition
Semaphores and P&V Operations

- Semaphores: a variable to indicate the # of pending wakeups
- **Down** operation (P):
  - Checks if a semaphore is > 0,
    - if so, it decrements the value and just continue
    - Otherwise, the process is put to sleep without completing the down for the moment
- **Up** operation (V)
  - Increments the value of the semaphore
    - if one or more processes are sleeping on the semaphore, one of them is chosen by the system (by random) and allowed to complete its down
- P & V operations are atomic, how to implement?
  - Single CPU: system calls, disabling interrupts temporarily
  - Multiple CPUs: TSL help

---

The Producer-consumer Problem w/ Semaphores

```c
#define N 100
typedef int semaphore;
s semaphore mutex = 1;
s semaphore empty = N;
s semaphore full = 0;

void producer(void)
{
    int item;
    while (TRUE) {
        item = produce_item();
        down(&empty);
        down(&mutex);
        mutex_term(item);
        put(mutex);
        up(0); /* increment count of full slots */
    }
}

void consumer(void)
{
    int item;
    while (TRUE) {
        down(&full);
        down(&mutex);
        item = remove_item();
        mutex_term(item);
        take(mutex);
        up(mutex);
        empty_term(item);
        up(mutex);
        /* do something with the item */
    }
}
```

Binary semaphores: is each process does a down before entering its critical region and an up just leaving it, mutual exclusion is achieved.
Mutexes

**Mutex:**
- A variable that can be in one of two states: unlocked or locked
- A simplified version of the semaphores

```assembly
mutex_lock:
    TSL REGISTER,MUTEX   | copy mutex to register and set mutex to 1
    CMP REGISTER,#0     | was mutex zero?
    JZ ok               | if it was zero, mutex was unlocked, so return
    CALL thread_yield   | mutex is busy; schedule another thread
    JMP mutex_lock      | try again later
ok: RET | return to caller; critical region entered

mutex_unlock:
    MOVE MUTEX,#0       | store a 0 in mutex
    RET | return to caller
```

Implementation of `mutex_lock` and `mutex_unlock` useful for user-space multi-threading

Mutexes – User-space Multi-threading

**What is a key difference between `mutex_lock` and `enter_region` in multi-threading and multi-processing?**

- For user-space multi-threading, a thread has to allow other thread to run and release the lock so as to enter its critical region, which is impossible with busy waiting `enter_region`

```assembly
enter_region:
    TSL REGISTER,LOCK   | copy lock to register and set lock to 1
    CMP REGISTER,#0     | was lock zero?
    JNE enter_region    | if it was non zero, lock was set, so loop
    RET | return to caller; critical region entered

leave_region:
    MOVE LOCK,#0        | store a 0 in lock
    RET | return to caller
```

Two processes entering and leaving a critical region using the TSL instruction
Re: The Producer-consumer Problem w/ Semaphores

```c
#define N 100 /* number of slots in the buffer */
typedef int semaphore; /* semaphores are a special kind of int */ semaphore mutex = 1; /* controls access to critical region */ semaphore empty = N; /* counts empty buffer slots */ semaphore full = 0; /* counts full buffer slots */

void producer(void) {
    int item;
    while (TRUE) {
        item = produce_item(); /* generate something to put in buffer */
        down(empty); /* decrement empty count */
        down(mutex); /* enter critical region */
        insert_item(item); /* put new item in buffer */
        up(mutex); /* leave critical region */
        up(&full); /* increment count of full slots */
    }
}

do the two downs in the producer’s code were reversed in order, so mutex was decremented before empty instead of after it?

void consumer(void) {
    int item;
    while (TRUE) {
        item = remove_item(); /* infinite loop */
        down(mutex); /* decrement full count */
        down(full); /* enter critical region */
        consume_item(item); /* take item from buffer */
        up(mutex); /* leave critical region */
        up(&empty); /* increment count of empty slots */
        do something with the item */
    }
}
```

Monitors (1)

- Monitor: a higher-level synchronization primitive
  - Only one process can be active in a monitor at any instant, with compiler’s help; thus, how about putting all the critical regions into monitor procedures for mutual exclusion?

```plaintext
monitor example
    integer i;
    condition c;

    procedure producer();
        ...
        end;

    procedure consumer();
        ...
        end;

end monitor:
```

But, how processes block when they cannot proceed?

Condition variables, and two operations: \texttt{wait()} and \texttt{signal()}

Example of a monitor
Monitors (2)

Monitors (3)

Outline of producer-consumer problem with monitors

- only one monitor procedure active at one time (a process doing `signal` must exit the monitor immediately); buffer has N slots

```java
public class ProducerConsumer {
    static final int N = 100; // constant giving the buffer size
    static producer p = new producer(); // instantiate a new producer thread
    static consumer c = new consumer(); // instantiate a new consumer thread
    static our_mon = new our_monitor(); // instantiate a new monitor
    public static void main(String arg[]) {
        p.start(); // start the producer thread
        c.start(); // start the consumer thread
    }
}

private class producer extends Thread {
    public run() { // run method contains the thread code
        while (true) {
            item = produce_item(); // producer loop
            mon.insert(item);
        }
    }
}

private class consumer extends Thread {
    public run() { // run method contains the thread code
        while (true) {
            item = mon.remove(); // consumer loop
            consume_item(item);
        }
    }
}
```

Solution to producer-consumer problem in Java (part 1)
Message Passing

```c
#define N 100

void producer(void) {
    int item;
    int message m; /* message buffer */
    while (TRUE) {
        item = produce_item(); /* generate something to put in buffer */
        build_message(&m, item);
        send(consumer, &m); /* construct a message to send */
        send_item_to_consumer(); /* send item to consumer */
    }
}

void consumer(void) {
    int item, i;
    int message m;
    for (i = 0; i < N; i++) send producer, &m; /* send N empties */
    while (TRUE) {
        receive(producer, &m); /* get message containing item */
        item = extract_item(&m); /* extract item from message */
        send(producer, &m); /* send back empty reply */
        consume_item(item); /* do something with the item */
    }
}

The producer-consumer problem with N messages
```

Barriers

- Use of a barrier (for programs operate in phases, neither enters the next phase until all are finished with the current phase)
  - processes approaching a barrier
  - all processes but one blocked at barrier
  - last process arrives, all are let through
Classic IPC Problems: Dining Philosophers (1)

° Philosophers eat/think
° Eating needs 2 forks
° Pick one fork at a time
° How to prevent deadlock & starvation
  • Deadlock: both are blocked on some resource
  • Starvation: both are running, but no progress made

The problem is useful for modeling processes that are competing for exclusive access to a limited number of resources, such as I/O devices

Dining Philosophers (2)

#define N 5 /* number of philosophers */

void philosopher(int i) /* i: philosopher number, from 0 to 4 */
{
    while (TRUE) {
        think(); /* philosopher is thinking */
        take_fork(i); /* take left fork */
        take_fork((i+1) % N); /* take right fork; % is modulo operator */
        eat(); /* yum-yum, spaghetti */
        put_fork(i); /* put left fork back on the table */
        put_fork((i+1) % N); /* put right fork back on the table */
    }
}

A non-solution to the dining philosophers problem

What happens if all philosophers pick up their left forks simultaneously?
Or, all wait for the same amount of time, then check if the right available?
What if random waiting, then check if the right fork available?
What performance if down and up on mutex before acquiring/replacing a fork?
Dining Philosophers (3)

```c
#define N 5 /* number of philosophers */
#define LEFT (i*N-1)%N /* number of i's left neighbor */
#define RIGHT (i+1)%N /* number of i's right neighbor */
#define THINKING 0 /* philosopher is thinking */
#define HUNGRY 1 /* philosopher is trying to get forks */
#define EATING 2 /* philosopher is eating */

typedef int semaphore; /* semaphores are a special kind of int */
int state[N]; /* array to keep track of everyone's state */
semaphore mutex = 1; /* mutual exclusion for critical regions */
semaphore s[N]; /* one semaphore per philosopher */

void philosopher(int i) /* i: philosopher number, from 0 to N-1 */
{
    while (TRUE) /* repeat forever */
    {
        think(); /* philosopher is thinking */
        take_forks(i); /* acquire two forks or block */
        eat(); /* yum-yum, spaghetti */
        put_forks(i); /* put both forks back on table */
    }
}
```

Solution to dining philosophers problem (part 1)

Dining Philosophers (4)

```c
void take_forks(int i) /* i: philosopher number, from 0 to N-1 */
{
    down(&mutex); /* enter critical region */
    state[i] = HUNGRY; /* record fact that philosopher i is hungry */
    test(i); /* try to acquire 2 forks */
    up(&mutex); /* exit critical region */
    down(&s[i]); /* block if forks were not acquired */
}

void put_forks(i) /* i: philosopher number, from 0 to N-1 */
{
    down(&mutex); /* enter critical region */
    state[i] = THINKING; /* philosopher has finished eating */
    test(LEFT); /* see if left neighbor can now eat */
    test(RIGHT); /* see if right neighbor can now eat */
    up(&mutex); /* exit critical region */
}

void test(i) /* i: philosopher number, from 0 to N-1 */
{
    if (state[] = HUNGRY && state[LLEFT] = EATING && state[RIGHT] = EATING) {}
    state[] = EATING;
    up(&s[i]);
}
```

Solution to dining philosophers problem (part 2)
The Readers and Writers Problem

typedef int semaphore; /* use your imagination */
semaphore mutex = 1; /* controls access to 'rc' */
semaphore db = 1; /* controls access to the database */
int rc = 0; /* # of processes reading or wanting to */

void reader(void) {
    while (TRUE) {
        down(&mutex); /* get exclusive access to 'rc' */
        rc = rc + 1;
        if (rc == 1) down(&db); /* if this is the first reader ... */
        read_data_base(); /* access the data */
        down(&mutex); /* get exclusive access to 'rc' */
        rc = rc - 1;
        if (rc == 0) up(&db); /* if this is the last reader ... */
        up(&mutex); /* release exclusive access to 'rc' */
        use_data_read(); /* noncritical region */
    }
}

void writer(void) {
    while (TRUE) {
        think_up_data(); /* noncritical region */
        down(&db); /* get exclusive access */
        write_data_base(); /* update the data */
        up(&db); /* release exclusive access */
    }
}

A solution to the readers and writers problem (database)

The Sleeping Barber Problem – Self-reading

[Diagram of the sleeping barber problem]
Scheduling: Introduction to Scheduling

- Scheduler: to make a choice whenever two or more processes are simultaneously in the ready state
  - Pick the right process to run
  - Makes efficient use of the CPU
  - More an issue to Servers than to PCs (clients)

- Process switching (context switching) is very costly
  - State of the current process must be saved; SP, PC, GPRs, etc
  - Memory map (e.g., memory reference bits in the page table)
  - MMU reloaded
  - Invalidating the memory cache

Scheduling: Process Behavior

- Bursts of CPU usage alternate with periods of I/O wait
  - a CPU-bound/CPU-intensive process
  - an I/O bound / I/O intensive process
    - I/O is when a process enters the blocked state waiting for an external device to complete its work
Scheduling: When to Schedule

- **When to schedule**
  - A process is created
  - A process exits/terminated
  - A process blocks on I/O, on a semaphore
  - An I/O interrupt occurs

- **Non-preemptive vs. preemptive scheduling**
  - What is necessary to enable preemptive scheduling?
  - It is possible in user-space multi-threading?

  **Clock interrupt**

---

Scheduling Algorithm Goals

**All systems**
- Fairness - giving each process a fair share of the CPU
- Policy enforcement - seeing that stated policy is carried out
- Balance - keeping all parts of the system busy

**Batch systems**
- Throughput - maximize jobs per hour
- Turnaround time - minimize time between submission and termination
- CPU utilization - keep the CPU busy all the time

**Interactive systems**
- Response time - respond to requests quickly
- Proportionality - meet users’ expectations

**Real-time systems**
- Meeting deadlines - avoid losing data
- Predictability - avoid quality degradation in multimedia systems
Scheduling in Batch Systems - FCFS

- FCFS is non-preemptive
  - A single queue of ready processes
  - When a the running process blocks, the first process on the queue is run next; when a blocked process becomes ready, like a newly arrived job, it is put on the end of the queue

- What is its greatest strength?
- What is its greatest disadvantage?
  - Example: there is one compute-bound process that runs for 1s at a time and then reads a disk block, many I/O-bound process that use little CPU but each have to perform 1000 disk reads to complete, what is the result? If a scheduler preempts the compute-bound process every 10ms, when the I/O-bound processes would finish?

FCFS Scheduling Example

- Consider four CPU-intensive jobs, A through D, arrive at a computer system at the same time. They have estimated running times of 8, 6, 2, and 4 minutes.

Determine the mean process turnaround time if using FCFS scheduling
Scheduling in Batch Systems - Shortest Job First

- Shortest job first is non-preemptive, optimal in terms of minimal average execution time
  - Assumption: all jobs are ready simultaneously; the run time of each job known in advance

- Shortest Remaining Time Next
  - A preemptive version of shortest job first, to allow new short jobs to get good service
  - Assumption: the run time of each job known in advance

In reality, how OS knows the run time of jobs in advance?

Shortest Job First Scheduling Example

- Consider four CPU-intensive jobs, A through D, arrive at a computer system at the same time. They have estimated running times of 8, 6, 2, and 4 minutes.

Determine the mean process response time if using shortest-job-first scheduling
Scheduling in Interactive Systems – Round-Robin

- Interactive systems: two-level scheduling
  - Round-Robin Scheduling
    - Pre-emptive, maintains a list of runnable processes
    - Quantum: a time interval for running
      - Extreme: processor sharing (per-instruction)
    - How to set the quantum
      - Cost of process/context switching vs. responsiveness

(a) list of runnable processes
(b) list of runnable processes after B uses up its quantum
**Round-Robin Scheduling Example**

- Consider four CPU-intensive jobs, A through D, arrive at a computer system at the same time. They have estimated running times of 8, 6, 2, and 4 minutes.

Determine the mean process response time if using R-R scheduling (assuming the computer system is multiprogramming and each job gets its fair share of CPU).

---

**Scheduling in Interactive Systems – Priority Scheduling**

- Priority Scheduling: takes external factors into account
  - Pre-emptive
  - How to avoid starvation due to strict priority?
    - WTP: time-dependent priority scheduling
  - hybrid of priority and RR scheduling in multiple queues

A scheduling algorithm with four priority classes
Consider four CPU-intensive jobs, A through D, arrive at a computer system at the same time. They have estimated running times of 8, 6, 2, and 4 minutes. Their priorities are 2, 4, 3, and 1, respectively.

Determine the mean process response time if using Priority scheduling

Scheduling in Interactive Systems – Alternatives

- Shortest Process Next
- Guaranteed Scheduling
- Lottery Scheduling
- Fair-Share Scheduling
Scheduling in Real-Time Systems

Schedulable real-time system

- Given
  - \( m \) periodic events
  - event \( i \) occurs within period \( P_i \) and requires \( C_i \) seconds
- Then the load can only be handled if
  \[
  \sum_{i=1}^{m} \frac{C_i}{P_i} \leq 1
  \]

- Example: a soft real-time system with three periodic events, with periods of 100, 200, and 500 ms, respectively. If these events require 50, 30, and 100 ms of CPU time per event, respective, the system is schedulable
  - Process/context switching overhead is often an issue though!

Policy versus Mechanism

- Separate what is allowed to be done with how it is done
  - a process knows which of its children threads are important and need priority
- Scheduling algorithm parameterized
  - mechanism in the kernel
- Parameters filled in by user processes
  - policy set by user process
Thread Scheduling – User-space Multi-threading

- Order in which threads run
- It can deploy an application-specific thread scheduler

Possible scheduling of user-level threads
- 50-msec process quantum
- Threads run 5 msec/CPU burst

Possible: A1, A2, A3, A1, A2, A3
Not possible: A1, B1, A2, B2, A3, B3

Thread Scheduling – Kernel-space Multi-threading

- 1. Kernel picks a thread

Possible scheduling of kernel-level threads
- 50-msec process quantum
- Threads run 5 msec/CPU burst

Possible: A1, A2, A3, A1, A2, A3
Also possible: A1, B1, A2, B2, A3, B3

How about performance?
Summary of Lecture 2

° Sequential process model
° Multi-threading: user-space vs. kernel-space
° IPC: semaphores, monitors, messages
  • Race conditions
  • Mutual exclusion
  • Critical regions
  • Classic IPC problems
° Scheduling
  • Process scheduling
  • Thread scheduling
° More reading: textbook 2.1 - 2.7