Mid-Term Review

(Operating-System Structures, Processes, Threads, Process Synchronization, CPU Scheduling, Deadlocks)

Summer 2018
Overview

• Operating-System Structures
• Processes
• Threads
• Process Synchronization
• CPU Scheduling
• Deadlocks
Operating-System Structures
System Calls

• Programming interface to the services provided by the OS
• Typically written in a high-level language (C or C++)
• Mostly accessed by programs via a high-level Application Programming Interface (API).

• Three most common APIs are
  • Win32 API for Windows,
  • POSIX API for UNIX, Linux and Mac OS X systems
  • Java API for the Java virtual machine (JVM)
System Calls – Passing Parameters

• Three general methods used to pass parameters to the OS
  • Pass the parameters in registers
    • In some cases, may be more parameters than registers
  • Parameters stored in a block, or table, and address of block passed as a parameter in a register
    • This approach taken by Linux and Solaris
  • Parameters placed, or pushed, onto the stack by the program and popped off the stack by the operating system
Operating System Structure - Layered

• The operating system is divided into a number of layers (levels), each built on top of lower layers. The bottom layer (layer 0), is the hardware; the highest (layer N) is the user interface.

• With modularity, layers are selected such that each uses functions (operations) and services of only lower-level layers.
1. What is the purpose of system calls?

2. Describe 3 general methods for passing parameters to the OS.

3. What is the purpose of system programs?
The experimental Synthesis operating system has an assembler incorporated in the kernel. To optimize system-call performance, the kernel assembles routines within kernel space to minimize the path that the system call must take through the kernel. This approach is the antithesis of the layered approach. Discuss the pros and cons of the Synthesis approach.
Processes
Process Concept

- **Multiple parts of a Process**
  - The program code, also called text section
  - **Stack** containing temporary data
    - Function parameters, return addresses, local variables
  - **Data section** containing global variables
  - **Heap** containing memory dynamically allocated during run time
As a process executes, it changes state:
- **New**: The process is being created
- **Running**: Instructions are being executed
- **Waiting**: The process is waiting for some event to occur
- **Ready**: The process is waiting to be assigned to a processor
- **Terminated**: The process has finished execution
Operations on Processes – Process Creation

- init (pid = 1)
  - login (pid = 8415)
    - bash (pid = 8416)
      - ps (pid = 9298)
    - khelper (pid = 6)
      - emacs (pid = 9204)
  - kthreadd (pid = 2)
    - pdflush (pid = 200)
  - sshd (pid = 3028)
    - sshd (pid = 3610)
      - tcsch (pid = 4005)
Operations on Processes – Process Termination

- The parent process may wait for termination of a child process by using the `wait()` system call. The call returns status information and the pid of the terminated process

  \[
  \text{pid} = \text{wait}(&\text{status});
  \]

- If no parent waiting (did not invoke `wait()`) process is a zombie

- If parent terminated without invoking `wait()`, process is an orphan
• What is the output at **LINE A**?

```c
#include <unistd.h>
int value = 5;
int main() {
    pid_t pid;
    pid = fork();
    if (pid == 0) {
        value += 15;
        return 0;
    }
    else if (pid > 0) {
        wait(NULL);
        printf("PARENT: value = %d", value); /*LINE A*/
        return 0;
    }
}
```
Processes
(Questions)

• Including the initial parent process, how many processes are created by this program?

```c
#include <unistd.h>

int main() {
    int i;

    for (i=0; i<4; i++)
        fork();

    return 0;
}
```
Processes
(Questions)

• When a process creates a new process using the `fork()` operation, **which of the following states is shared** between the parent process and the child process?
  • Stack
  • Heap
  • Shared memory segments
Threads
Overview

single-threaded process

multithreaded process
Overview

• **Process** creation is *heavy-weight* while **thread** creation is *light-weight*

• **Kernels** are generally **multithreaded**

• *Example for threads:* Multiple tasks within the application can be implemented by separate threads. In web-browser:
  • Update display
  • Fetch data
  • Spell checking
  • Answer a network request
Multithreaded Models: One-to-One

- Each user-level thread maps to unique kernel thread
- Creating a user-level thread creates a kernel thread
- More concurrency than many-to-one
- Number of threads per process sometimes restricted due to overhead
- Examples
  - Windows
  - Linux
  - Solaris 9 and later
Multithreaded Models: Many-to-Many

- Allows many user-level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Examples
  - Solaris prior to version 9
Threads
(Questions)

• How many unique processes are created?
• How many unique threads are created?

```c
pid_t pid;
pid = fork();
pf (pid == 0) { /* child process */
    fork();
    thread_create(...);
}
fork();
```
Threads (Questions)

• Which of the following components of program state are shared across threads in a multithreaded process?
  • Register values
  • Heap memory
  • Global variables
  • Stack memory

• Is it possible to have concurrency but not parallelism? Explain.
Process Synchronization
Background

• Concurrent access to shared data may result in data inconsistency

• Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
Critical Section Problem – 3
Important Guarantees to be provided

1. **Mutual Exclusion** - If process $P_i$ is executing in its critical section, then **no other processes** can be executing in their critical sections

2. **Progress** - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the **selection** of the processes that will enter the critical section next **cannot be postponed indefinitely**

3. **Bounded Waiting** - A **bound must exist** on the number of times that other processes are allowed to enter their critical sections **after a process has made a request** to enter its critical section and before that request is granted
Synchronization Hardware

• Many systems provide **hardware support** for implementing the critical section code.

• All solutions below based on idea of **locking**
  • Protecting critical regions via locks

• Uniprocessors – could **disable interrupts**
  • OSes using this not scalable to multiprocessor systems

• Modern machines provide special **atomic hardware instructions**
  • Atomic = non-interruptible
  • Either test memory word and set value – `test_and_set()`
  • Or swap contents of two memory words – `compare_and_swap()`
Mutex Locks

• **Previous solutions are complicated** and generally inaccessible to application programmers

• Simplest OS-level approach called **Mutex Locks**

• Protect a critical section by first **acquire()** a lock then **release()** the lock
  • Boolean variable indicating if lock is available or not

• Calls to **acquire()** and **release()** must be atomic
  • Usually implemented via hardware atomic instructions

• But this solution requires **busy waiting**
  • This lock therefore called a **spinlock**
Semaphores

• Synchronization tool that provides more sophisticated ways (than Mutex locks) for process to synchronize their activities.

• Semaphore \( S \) – integer variable

• Can only be accessed via two indivisible (atomic) operations: wait() and signal(). Originally called \( P() \) and \( V() \)

• Definition of the wait() operation
  
```c
  wait(S) {
    while (S <= 0) // busy wait
      S--; 
  }  
```

• Definition of the signal() operation
  
```c
  signal(S) {
    S++; 
  }  
```
Process Synchronization (Questions)

• What is the meaning of the term busy waiting? Can busy waiting be avoided altogether? Explain your answer.

• Show that, if the `wait()` and `signal()` semaphore operations are not executed atomically, then mutual exclusion may be violated.
CPU Scheduling
CPU Scheduler

- **Short-term scheduler** selects from among the processes in ready queue, and allocates the CPU to one of them
- CPU scheduling decisions may take place when a process:
  1. Switches from *running* to *waiting* state
  2. Switches from *running* to *ready* state
  3. Switches from *waiting* to *ready*
  4. Terminates
Scheduling Criteria

1. **CPU utilization** – keep the CPU as busy as possible
2. **Throughput** – # of processes that complete their execution per time unit
3. **Turnaround time** – amount of time to execute a particular process
4. **Waiting time** – amount of time a process has been waiting in the ready queue
5. **Response time** – amount of time it takes from when a request was submitted until the first response is produced
Scheduling Algorithms – Example of Round-Robin

<table>
<thead>
<tr>
<th>Process</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>(P_1)</td>
<td>24</td>
</tr>
<tr>
<td>(P_2)</td>
<td>3</td>
</tr>
<tr>
<td>(P_3)</td>
<td>3</td>
</tr>
</tbody>
</table>

• The Gantt chart is:

• Typically, higher average turnaround than SJF, but better response
• \(q\) should be large compared to context switch time
• \(q\) usually 10ms to 100ms, context switch < 10 usec
Real-Time CPU Scheduling: Priority-based Scheduling

- For **real-time scheduling**, scheduler must support **preemptive, priority-based** scheduling
  - But only guarantees soft real-time
- For **hard real-time** must also provide ability to meet deadlines
- Processes have new characteristics: **periodic** ones require CPU at constant intervals
  - Has processing time $t$, deadline $d$, period $p$
  - $0 \leq t \leq d \leq p$
  - **Rate** of periodic task is $1/p$
Real-Time CPU Scheduling: Rate Monotonic Scheduling

- A priority is assigned based on the inverse of its period
- Shorter periods = higher priority;
- Longer periods = lower priority
- \( P_1 \) is assigned a higher priority than \( P_2 \).

<table>
<thead>
<tr>
<th>Event</th>
<th>Period</th>
<th>Deadline</th>
</tr>
</thead>
<tbody>
<tr>
<td>( P_1 )</td>
<td>50</td>
<td>before ( t_1 ) = 20</td>
</tr>
<tr>
<td>( P_2 )</td>
<td>100</td>
<td>before ( t_2 ) = 35</td>
</tr>
</tbody>
</table>

Deadline is to complete before next period.
Real-Time CPU Scheduling: Missed Deadlines with Rate Monotonic Scheduling

P1 = 50
P2 = 80
t1 = 25
t2 = 35
Deadline is to complete before next period

There is no pre-emption in Rate Monotonic Scheduling
Real-Time CPU Scheduling: Earliest Deadline First Scheduling (EDF)

- Priorities are assigned according to deadlines:
  - The earlier the deadline, the higher the priority
  - The later the deadline, the lower the priority

\[ P_1 = 50 \]
\[ P_2 = 80 \]
\[ t_1 = 25 \]
\[ t_2 = 35 \]
CPU Scheduling (Questions)

• Discuss how the following pairs of scheduling criteria conflict in certain settings.
  • CPU utilization and response time.
  • Average turnaround time and maximum waiting time.
  • I/O device utilization and CPU utilization.

• Explain the difference in how much the following scheduling algorithms discriminate in favor of short processes:
  • FCFS
  • RR
  • Multilevel feedback queues.
CPU Scheduling (Questions)

• The nice command is used to set the nice value of a process on Linux. Explain why some systems may allow any user to assign a process a nice value \( \geq 0 \) yet allow only the root user to assign nice values \( <0 \).
The following processes are being scheduled using a preemptive, round-robin scheduling algorithm. Each process is assigned a numerical priority, with a higher number indicating higher relative priority. The system also has an idle task. This task had priority 0 and scheduled whenever there is no other processes to run. The length of time quantum is 10 units. If a process is preempted by a higher-priority process, the preempted process is placed at the end of the queue.
CPU Scheduling

• Show the scheduling order of the process using Gantt chart.
• What is the turnaround time for each process?
• What is the waiting time for each process?
• What is the CPU utilization rate?

<table>
<thead>
<tr>
<th>Thread</th>
<th>Priority</th>
<th>Burst</th>
<th>Arrival</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>40</td>
<td>20</td>
<td>0</td>
</tr>
<tr>
<td>P2</td>
<td>30</td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>P3</td>
<td>30</td>
<td>25</td>
<td>30</td>
</tr>
<tr>
<td>P4</td>
<td>35</td>
<td>15</td>
<td>60</td>
</tr>
<tr>
<td>P5</td>
<td>5</td>
<td>10</td>
<td>100</td>
</tr>
<tr>
<td>P6</td>
<td>10</td>
<td>10</td>
<td>105</td>
</tr>
</tbody>
</table>
Deadlocks
Deadlock Characterization

Deadlock can arise if four conditions hold simultaneously.

- **Mutual exclusion**: only one process at a time can use a resource
- **Hold and wait**: a process holding at least one resource is waiting to acquire additional resources held by other processes
- **No preemption**: a resource can be released only voluntarily by the process holding it, after that process has completed its task
- **Circular wait**: there exists a set \( \{P_0, P_1, \ldots, P_n\} \) of waiting processes such that \( P_0 \) is waiting for a resource that is held by \( P_1 \), \( P_1 \) is waiting for a resource that is held by \( P_2 \), \ldots, \( P_{n-1} \) is waiting for a resource that is held by \( P_n \), and \( P_n \) is waiting for a resource that is held by \( P_0 \).
Deadlock Characterization

Deadlock can arise if **four conditions** hold simultaneously.

- **Mutual exclusion**: only one process at a time can use a resource
- **Hold and wait**: a process holding at least one resource is waiting to acquire additional resources held by other processes
- **No preemption**: a resource can be released only voluntarily by the process holding it, after that process has completed its task
- **Circular wait**: there exists a set \{P_0, P_1, ..., P_n\} of waiting processes such that \(P_0\) is waiting for a resource that is held by \(P_1\), \(P_1\) is waiting for a resource that is held by \(P_2\), ..., \(P_{n-1}\) is waiting for a resource that is held by \(P_n\), and \(P_n\) is waiting for a resource that is held by \(P_0\).
Deadlock Characterization: Resource-Allocation Graph with A Deadlock
Deadlock Characterization: Graph with a Cycle But No Deadlock
Deadlock Characterization : Basic Facts

• If graph contains **no cycles** ⇒ no deadlock

• If graph contains **a cycle** ⇒
  • if only **one instance per resource type**, then deadlock
  • if **several instances per resource type**, possibility of deadlock
Deadlocks
(Questions)

• Show that the **four necessary conditions for deadlocks** hold in this example.

• State a **simple rule for avoiding deadlocks** in this system.
Deadlocks
(Questions)

• Is it possible to have a deadlock involving only one single-threaded process? Explain your answer.

• The following program doesn’t always lead to deadlock. Describe what role CPU scheduler plays and how it can contribute to deadlock in this program.
Deadlock Prevention:
Deadlock Example

/* thread one runs in this function */

void *do_work_one(void *param)
{
    pthread_mutex_lock(&first_mutex);
    pthread_mutex_lock(&second_mutex);
    /** * Do some work */
    pthread_mutex_unlock(&second_mutex);
    pthread_mutex_unlock(&first_mutex);
    pthread_exit(0);
}

/* thread two runs in this function */

void *do_work_two(void *param)
{
    pthread_mutex_lock(&second_mutex);
    pthread_mutex_lock(&first_mutex);
    /** * Do some work */
    pthread_mutex_unlock(&first_mutex);
    pthread_mutex_unlock(&second_mutex);
    pthread_exit(0);
}
Good Luck!