Coe628 Midterm Study Guide (2017)

The Questions

1. Multi-core state diagram

A single CPU (core) has a state transition diagram indicating the possible process state changes (RUN-READY-BLOCKED). Suppose there are 4 cores. Some process is running on each core: call these states RUN-1, RUN-2, RUN-3 and RUN-4.

What state transitions are allowable?

2. Amdahl's Law

An algorithm is 20% serial; the rest can be run in parallel. What is the maximum speedup if 5 processors can be used?

3. Threads vs. Processes

a) What are some pros and cons of processes verses threads?

b) What is the main advantage and main disadvantage to user threads vs. Kernel threads.

c) Give 3 characteristics common to both threads and processes.

4. Deadlock avoidance

TaskA and TaskB both need both resources, P and Q, in order to do some work. Each Task can acquire a resource with acquireP() or acquireQ(). The resources cannot be shared. If a Task tries to acquire a resource that is already in use, it blocks. When a Task no longer needs a resource it owns it can release it with releaseP() or releaseQ().

a) Each task runs its own code and uses the functions acquireP(), acquireQ(), doWork(), releaseP() and releaseQ(). Each thread releases its resources after invokes “doWork()”; the resources are released in the opposite order they were obtained.

Show a sequence of operations for each thread so that deadlock is impossible.

TaskA **TaskB**

b) Show a sequence of operations for each task such that deadlock is possible. Then show a specific sequence where deadlock is avoided (task switching takes place at just the right times) and another specific sequence that does result in deadlock.

ThreadA **ThreadB (with sequence that does NOT deadlock)**

5. Fork/exec/shell parsing

1. The following program is run. Assume that it's process ID is 1000 and that the next processes created will have process IDs 1001, 1002, 1003, etc. What is the output from the program?

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main() {
 int i = 3;
 int child;
 int status;
 if((child = fork()) == 0) {
    i = i + 2;
    printf("X %d %d\n", child, i);
    if((child = fork()) == 0) {
     printf("Z %d %d\n", child, ++i);
     return -2;
    }
    wait(&status);
    printf("A %d %d\n", child, i);
    return -1;
 }
 wait(&status);
```

```
 printf("Y %d %d\n", child, i);
 return 0;
}
```

The Answers

1. Multi-core state diagram

The following transitions are legal:

- RUN-(1 or 2 or 3 or 4) TO BLOCKED
- RUN-(1 or 2 or 3 or 4) TO READY
- READY TO RUN-(1 or 2 or 3 or 4)
- BLOCKED TO READY

2. Amdahl's Law

ANSWER (BASIC): The speed up factor is 2.8.

3. Threads vs. Processes

a) Processes have independent memory spaces; threads share global memory. Thread creation is simpler than process creation.

b) Kernel threads can be blocked (from a system call) without blocking other threads. If a user-level thread blocks from a system call, all threads associated with the process will also be blocked.

c) Both threads and processes can be in states such as RUN-READY-BLOCKED. Both Threads and Processes have associated tables which give the state, saved registers, ids etc. Of each entry.

4. Deadlock avoidance

TaskA and TaskB both need both resources, P and Q, in order to do some work. Each Task can acquire a resource with acquireP() or acquireQ(). The resources cannot be shared. If a Task tries to acquire a resource that is already in use, it blocks. When a Task no longer needs a resource it owns it can release it with releaseP() or releaseQ().

a) Each task runs its own code and uses the functions acquireP(), acquireQ(), doWork(), releaseP() and releaseQ(). Each thread releases its resources after invokes "doWork()"; the resources are released in the opposite order they were obtained.

Show a sequence of operations for each thread so that deadlock is impossible.

TaskA **TaskB**

b) Show a sequence of operations for each task such that deadlock is possible. Then show a specific sequence where deadlock is avoided (task switching takes place at just the right times) and another specific sequence that does result in deadlock.

ThreadA **ThreadB (with sequence that does NOT**

5. Fork/exec/shell parsing

1. The following program is run. Assume that it's process ID is 1000 and that the next processes created will have process IDs 1001, 1002, 1003, etc. What is the output from the program?

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main() {
 int i = 3;
 int child;
 int status;
```

```
 if((child = fork()) == 0) {
    i = i + 2;
    printf("X %d %d\n", child, i);
    if((child = fork()) == 0) {
     printf("Z %d %d\n", child, ++i);
     return -2;
    }
    wait(&status);
    printf("A %d %d\n", child, i);
    return -1;
 }
 wait(&status);
 printf("Y %d %d\n", child, i);
 return 0;
}
```

The Answers

1. Multi-core state diagram

The following transitions are legal:

- RUN-(1 or 2 or 3 or 4) TO BLOCKED
- RUN-(1 or 2 or 3 or 4) TO READY
- READY TO RUN-(1 or 2 or 3 or 4)
- BLOCKED TO READY

2. Amdahl's Law

ANSWER (BASIC): The speed up factor is 2.8.

3. Threads vs. Processes

a) Processes have independent memory spaces; threads share global memory. Thread creation is

simpler than process creation.

b) Kernel threads can be blocked (from a system call) without blocking other threads. If a user-level thread blocks from a system call, all threads associated with the process will also be blocked.

c) Both threads and processes can be in states such as RUN-READY-BLOCKED. Both Threads and Processes have associated tables which give the state, saved registers, ids etc. Of each entry.

4. Deadlock avoidance

If each thread acquires the resources in OPPOSITE order, deadlock is POSSIBLE (but NOT inevitable).

If acquired in the same order, no deadlock can occur.

5. Fork/exec/shell parsing

```
X 0 5
Z 0 6
A 1002 5
Y 1001 3
```