CS2106 – Introduction to Operating Systems AY24/25 Semester 2, Y2S2 Tutorials

Sim Ray En Ryan (A0273780X, e1123183@u.nus.edu)

December 3, 2025

Contents

1 Tutorial 1: MIPS Assembly Revision and Stack Frames

1.1 MIPS Assembly Revision

1.1.1 C to MIPS

Write the following C code in MIPS:

```
int f(int x, y) {
    return 2 * (x + y);
}
int a = 3, b = 4, y;

int main() {
    y = f(a, b);
}
```

- la t0, a
- lw a0, 0 (t0) // Get value of a
- la t0, b
- lw a1, 0 (t0) // Get value of b
- jal f
- la t0 y
- sw v0, 0 (t0) // Store value of y
- System call to exit
- f:
- add t1, a0, a1
- sll v0, t1, 1
- jr ra

1.1.2 C to MIPS using stack and frame pointers

Repeat the above using stack to pass arguments instead of a0, a1, and v0.

- addi fp, sp, 0 // store sp
- addi sp, sp, 8 // reserve space for 2 int
- la t0, a
- lw t0, 0 (t0)
- sw t0, 0 (fp) // store first int
- la t0, b
- lw t0, 0 (t0)
- sw t0, 4 (fp) // store second int
- jal f
- lw t1, 0 (fp)
- la t0 y
- sw t1, 0 (t0)
- addi sp, sp, -8 // Restore stack pointer
- System call to exit
- f:
- lw t0, 0 (fp)
- lw t1, 4 (fp)
- add t2, t0, t1
- sll v0, t2, 1
- sw v0, 0 (fp)
- jr ra

1.2 Stack Frames

1.2.1 Recursion and nested function calls

1.2.2 Proper Stack Frame

Can your approach to passing parameters and calling functions in Questions 1 and 2 above work for recursive or even nested function calls? Explain why or why not.

It will not work, since ra is not saved. The original ra will be lost.

1.2.3 Restoration

- Store fp
- Copy new sp
- Reserve 20 bytes, and change offsets

1.2.4 Saved Registers

In Question 4 the callee saved registers it intends to use onto the stack and restores them after that. What would happen if the callee does not do that? Why don't we do the same thing for main

The callee does not know what registers the caller uses, and does not want to overwrite the contents of those. The main function is invoked by OS, and OS will save the registers it needs. The compiler also generates extra code for main, especially if it takes cmd arguments.

1.2.5 Retrieval of $ra

Explain why, in step 7 of the callee, we retrieve $rafromthestackbeforedoingjr$ra. Why can't we just do jr ra directly?

ra could have been overwritten by nested functions, which will cause a loop.

1.3 Process Memory

1.3.1 Storage of processes

- Code and Text are stored in ROM
- Global Variables are stored in Data
- Dynamically Allocated Memory are found in the Heap
- Variables, Pointers, and other results are found on the Stack

2 Tutorial 2: Process Abstraction in Unix

2.1 Process Creation Recap - Taken from AY18/19 S1 Midterms

2.1.1 a

False. Q does always terminate before P, but Q has to wait for R to terminate

2.1.2 b

False, P will not wait, and Q still needs to wait for R

2.1.3 c

True, Q does always terminate before P, even if Q is a new executable.

2.1.4 d

False, it will terminate. The wait will return immediately when there is no child.

2.2 Behavior of Fork

2.2.1 Q2a.

What is the difference between the 3 variables: dataX, dataY and the memory location pointed by dataZptr?

dataX is a global variable, and is in the Text section. dataY is a variable, and is on the stack, and dataZptr is dynamically allocated, and on the heap

2.2.2 Q2b.

Focusing on the messages generated by second phase (they are prefixed with either "*" and ""), what can you say about the behavior of the fork() system call?

After forking, each process has independent memory space.

2.2.3 Q2c.

Using the messages seen on your system, draw a process tree to represent the processes generated. Use the process tree to explain the values printed by the child processes.

In the first phase, one process is alive. In the second, it has spawned a child. In the third, both will spawn a child.

2.2.4 Q2d.

Do you think it is possible to get different ordering between the output messages, why?

Yes, once different processes are spawned, they are independently run by the OS.

2.2.5 Q2e.

Can you point how which pair(s) of messages can never swap places? i.e. their relative order is always the same?

Messages that come from the same process can will always be in the same order. Messages from the first phase will precede all other messages. Note that a child's message can prefix a parents message, and messages in the same phase do not necessarily have to precde the next phase.

2.2.6 Q2f.

The first process and the second process spawned by the first process will execute first, followed by the first process spawned and the process that spawns. This is assuming execution completes in under 5 seconds.

2.2.7 Q2g.

THe first phase, then the second phase of the process spawned, then the process that spawns, then the second spawn of the first process.

2.3 Q3. Parallel Computation

The parallel solution will not have any difference if there is only one processor, of it the cost of spawning a new processor is more than what is gained from parallel execution. Changing wait to waitpid means that main processes has to wait for child processes in the same order (such as in the same way as the input data)

2.4 Q4. Process Creation

Given the following full program, give and explain the execution output. The source code FF.c is also given for your own test.

It will print twice, as fork() duplicates entire memory region, including all previous calls.

2.5 Q5. Process Creation

2.5.1 a. A possible final set of printed messages

PID 100 x=10 y=102 PID 102 x=9 y=0 PID 101 x=9 y=103 PID 103 x=8 y=0

2.5.2 b. An impossible final set of printed messages

PID 100 x=10 y=102 PID 101 x=9 y=0 PID 102 x=9 y=103 PID 103 x=8 y=0

PID 102 does not spawn 103, so this is wrong.

2.6 Q6. Parent-Child Synchronization

2.6.1 a, wait()

Child X is done always appears before one child exited

2.6.2 b. waitpid()

Parent one child existed only prints after processes terminate.

3 Tutorial 3: Process Scheduling

3.1 Putting it together

3.1.1 D = 1

```
[120]: Step 0
[120]: Step 1
[120]: Step 2
[120]: Step 3
[120]: Step 4
[121]: Step 0
[121]: Step 1
[121]: Step 2
[121]: Step 3
[121]: Step 4
[121] Child Done!
[120] Parent Done!
```

Delay is based on D. When D is very short, the parent process can finish before the child does, since it is shorter than the time quantum.

3.1.2 D = 100 million

```
[122]: Step 0
[123]: Step 0
[123]: Step 1
[122]: Step 1
[123]: Step 2
[122]: Step 2
[123]: Step 3
[122]: Step 3
[123]: Step 4
[122]: Step 4
[123] Child Done!
[122] Parent Done!
```

The delay here is very long, and both parent and child will be executing interleavedly. This is because D is now larger than the time quantum. If D is increased further, it may take multiple time quantum for each iteration

3.1.3 Smallest D for interleaved pattern

About 115000. Execution time of for loop and print statement, which now represents the time quantum.

3.2 Walking through scheduling algorithms

3.2.1 FCFS Scheduling Chart

CPU: AAABCCAAAAB__BB_

3.2.2 Turnaround and waiting time for A, B, and C

- A: 10, 2
- B: 15, 6
- C: 3, 1

3.2.3 Round Robin Scheduling Chart

CPU: AABACCBAABBA

3.2.4 Response Time

- A: 0
- B: 2
- C: 1

3.3 Adapted from AY1617S1 Midterm – MLFQ

CPU: AABABAABA

3.4 Adapted from Midterm 1516/S1 – Understanding of Scheduler

3.4.1 Psuedocode for Round-Robin

- PrevTask = TempTask
- while ReadyQ !isEmpty,
- RunningTask = ReadyQ.dequeue()
- SwitchContext(TempTask, RunningTask)
- RunningTask.TQLeft -= 1
- ReadyQ.enqueue(RunningTask)

3.4.2 Blocking of I/O and events

Can have a try, except block added that will continue the loop and add it to another Queue. This other queue will then add it back to ReadyQ once the task is ready. For IO, the process needs to make an IO call, and the OS can intercept those events instead.

3.5 MLFQ

3.5.1 Change of Heart

When the process finally reaches the I/O phase, it will have such a low priority that it will likely starve.

Rule 2, Timely Boost, will fix this issue.

3.5.2 Gaming the System

The process's priority will never decrease and will likely complete before most other processes in the queue.
Rule 1, Accounting matters, will fix this issue.

3.6 Predicting CPU Time

In the formula, alpha represents the weight of the immediate past value, while 1-alpha represents the past history.

- 10, 9, 0.1111
- 9.5, 8, 0.1875
- 8.75, 8, 0.09375
- 8.375, 7, 0.196429
- 7.6875, 6, 0.28125
- Average Error: 0.173986
- 10, 8, 0.25
- 9, 14
- 11.5, 3
- 7.25, 18
- 12.625, 2
- Average Error: A lot

3.7 Scheduler Case Study: Linux

3.7.1 Benefit of array of 140 Linked Lists

This changes the time complexity of finding the highest priority task from O(n) to O(1).

3.7.2 Two sets of tasks

This prevents starvation in that no task that has already ran (in the expired tasks) can become an active task. The duration to run all tasks in “active” is known as time epoch in Linux

3.7.3 Variable Time Slicing

This will mitigate the effect of starvation by higher-priority tasks. High priority tasks may also be shorter in nature. So, it balances out.

3.7.4 Bonus and Penalty

This is so that functions that may have a very long I/O phase such as in earlier can actually be run, while any function trying to block before the time quantum does not actually gain benefits. This also allows the OS to distinguish between computation intensive and IO intensive processes, and help those that switch between the two.

4 Tutorial 4: Process Scheduling and IPC

4.1 Evaluating Scheduling Algorithms

4.1.1 FIFO shortest average response time

Under what conditions does FCFS (FIFO) scheduling result in the shortest possible average response time?

When there is not a process with very long waiting time in the start of the queue: When the processes are sorted from shortest to longest execution time (or condescending).

4.1.2 RR same as FIFO

Under what conditions does round-robin (RR) scheduling behave identically to FIFO?

When all processes complete before the time quantum.

4.1.3 RR worse than FIFO

Under what conditions does RR scheduling perform poorly compared to FIFO?

Ceteris Paribus, when the tasks take longer than the time-quantum, as it will involve many context switches and overhead by the OS.

4.1.4 Time quantum and RR

Does reducing the time quantum for RR scheduling help or hurt its performance relative to FIFO, and why?

- Lowers performance: Lower time quantum increases amount of CPU time spent on context switching, reducing throughput and increasing average turnaround time, and overall time.
- Increases performance: Lower time quantum reduces time for task to get CPU, increasing responsiveness.

4.1.5 CPU or I/O

Do you think a CPU-bound (CPU-intensive) process should be given a higher priority for I/O than an I/O-bound process? Why?

- Yes: This decreases overall turnaround time
- No: The responsiveness of the IO should be maximised

4.2 Shared Memory

4.2.1 Expected final value

Let n = 100, nChild = 1. The expected final value is 200.

4.2.2 Running multiple times with varied values of n and nChild

Let n = 10000, nChild = 10. Then, 11 * 10000 = 110000 (which is correct) But running it multiple times, it will range.

4.2.3 Binding to a single core

After fixing it to a single processor with taskset -c 5 ./a.out 10000 100, it is 110000 all the time. Actually, it is still possible on single-cored processors as the first process might be swapped out due to the time quantum. However, n needs to be way higher, such as with 10 million.

4.3 Protecting Shared Memory

Modify shm_protected.c to create a shared memory region, and a critical section of code through busy waiting. This will result in a longer execution time, but also prevent simultaneous access.

5 Tutorial 5: Synchronization

Since synchronization is important to both multi-threaded and multi-process programs, the word task will be used.

5.1 Race Conditions

- Task A: x++; x++;
- Task B: x = 2*x;

Possible values:

- 0: Load x = 0 for multiplication, run both addition, return 0 from multiplication
- 1: Both x++ and x=2*x load x = 0 at the same time, 0 is returned, then add 1
- 2: B runs before A
- 3: + * + 0 1 2 1
- 4: * + + 0 1 1 2

Since there are 4 relevant atomic operations in A and 2 in B, there are 5 slots to place the two B, with half being in the wrong order. Thus, there is a total of 15 methods for 5 outcomes.

5.2 Critical Selection

Yes, there is guaranteed to only be one process running on that core. However:

- This does not work on multi-core processes.
- Disabling interrupts requires kernel level permissions
- It will also break many scheduling algorithms
- High priority and urgent tasks will also not be able to preempt, even if they do not use the same shared region
- If the program hangs, it will hang forever

5.3 Atomic Increment

```
int atomic_increment(int* t) {
    do {
        int temp = *t;
    } while (!_sync_bool_compare_and_swap(t, temp, temp + 1);
    return temp + 1;
}
```

5.4 Locks without mutexes or semaphores

Use the pipe, and have a null byte in ??

6 Tutorial 6: Synchronization II

6.1 Semaphores and three tasks

Consider three concurrently executing tasks using two semaphores S1 and S2 and a shared variable x. Assume S1 has been initialized to 1, while S2 has been initialized to 0. What are the possible values of the global variable x, initialized to 0, after all three tasks have terminated?

A	B	C

P(S2);
P(S1);
$x = x \times 2;$
V(S1);
	P(S1);
	$x = x \times x;$
	V(S1);
		P(S1);
		$x = x + 3;$
		V(S2);
		V(S1);

**Note: P(), V() are a common alternative name for Wait() and Signal() respectively.*

A must occur after C.

6.1.1 B, C, A

$x = 0 * 0 = 0$
$x = x + 3 = 3$
$x = x * 2 = 6$

6.1.2 C, B, A

$x = x + 3 = 3$
$x = x * x = 9$
$x = x * 2 = 18$

6.1.3 C, A, B

$x = x + 3 = 3$
$x = x * 2 = 6$
$x = x * x = 36$

6.2 Sempahores

In cooperating concurrent tasks, sometimes we need to ensure that all N tasks reach a certain point in code before proceeding. This specific synchronization mechanism is commonly known as a **barrier**. Example usage:

```
//some code

Barrier( N );   // The first N-1 tasks reaching this point
```

```
                // will be blocked.
                // The arrival of the Nth task will release
                // all N tasks.

// Code here only gets executed after all N processes
// reached the barrier above.
```

Use semaphores to implement a **one-time use Barrier()** function **without using any form of loops**. Remember to indicate the variable declarations clearly.

```
mutex = Semaphore 1
barrier = Semaphore 0
count = 0

void Barrier (N)
    Wait Mutex
    count++
    if count == N
        Signal barrier
    Signal Mutex

    Wait barrier
    Signal barrier
```

6.3 Deadlocks: The Stubborn Villagers Problem

A village has a long but narrow bridge that does not allow people crossing in opposite directions to pass by each other. All villagers are very stubborn and will refuse to back off if they meet another person on the bridge coming from the opposite direction.

6.3.1 (a) Explain how the behavior of the villagers can lead to a deadlock

When two people arrive at opposite sides at the same time.

6.3.2 (b) Analyze the correctness of the following solution and identify the problems, if any

```
Semaphore sem = 1;

void enter_bridge()
{
    sem.wait();
}

void exit_bridge()
{
    sem.signal();
}
```

Multiple people cannot cross the bridge in the same direction.

6.3.3 (c) Modify the above solution to support multiple people crossing the bridge in the same direction. You are allowed to use a single shared variable and a single semaphore

```
Semaphore sem = 1;
int currentDirection = 0

void enter_bridge(int direction)
{
    if direction and currentDirection have same sign, return
    else, sem.wait(), currentDirection
}

void exit_bridge()
{
    sem.signal();
}
```

6.3.4 (d) What is the problem with the solution in (c)?

People on the other side can be starved from crossing the bridge if people keep crossing.

6.4 General Semaphore: Implementing Using Binary Semaphores

We mentioned that a general semaphore ($S > 1$) can be implemented using a **binary semaphore** ($S == 0$ or $S == 1$). Consider the following attempt:

```
int count = <initially: any non-negative integer>;
Semaphore mutex = 1;   // binary semaphore
Semaphore queue = 0;   // binary semaphore, for blocking tasks
```

GeneralWait()	GeneralSignal()

`wait( mutex );`
`count = count - 1;`
`if (count < 0) {`
`    signal( mutex );`
`    wait( queue )`
`} else {`
`    signal( mutex );`
`}`
	`wait( mutex );`
	`count = count + 1;`
	`if (count <= 0) {`
	`    signal( queue );`
	`}`
	`signal( mutex );`

Note: For ease of discussion, we allow the count to go negative in order to keep track of the number of tasks blocked on the queue.

6.4.1 (a) The solution is very close, but unfortunately can still have undefined behavior in some execution scenarios.

Give one such execution scenario to illustrate the issue.
(Hint: Binary semaphore works only when its value $S = 0$ or $S = 1$.)

6.4.2 (b) [Challenge] Correct the attempt.

Note that you only need very small changes to the two functions.

6.5 Synchronization Problem – Dining Philosophers

Our philosophers in the lecture are all left-handed (they pick up the left chopstick first). If we force **one of them** to be a right-hander, i.e., pick up the right chopstick before the left, then it is claimed that the philosophers can eat without explicit synchronization.

Do you think this is a **deadlock-free solution** to the dining philosopher problem? You can support your claim informally (i.e., no need for a formal proof).

Yes, since there is no more cyclic dependency.